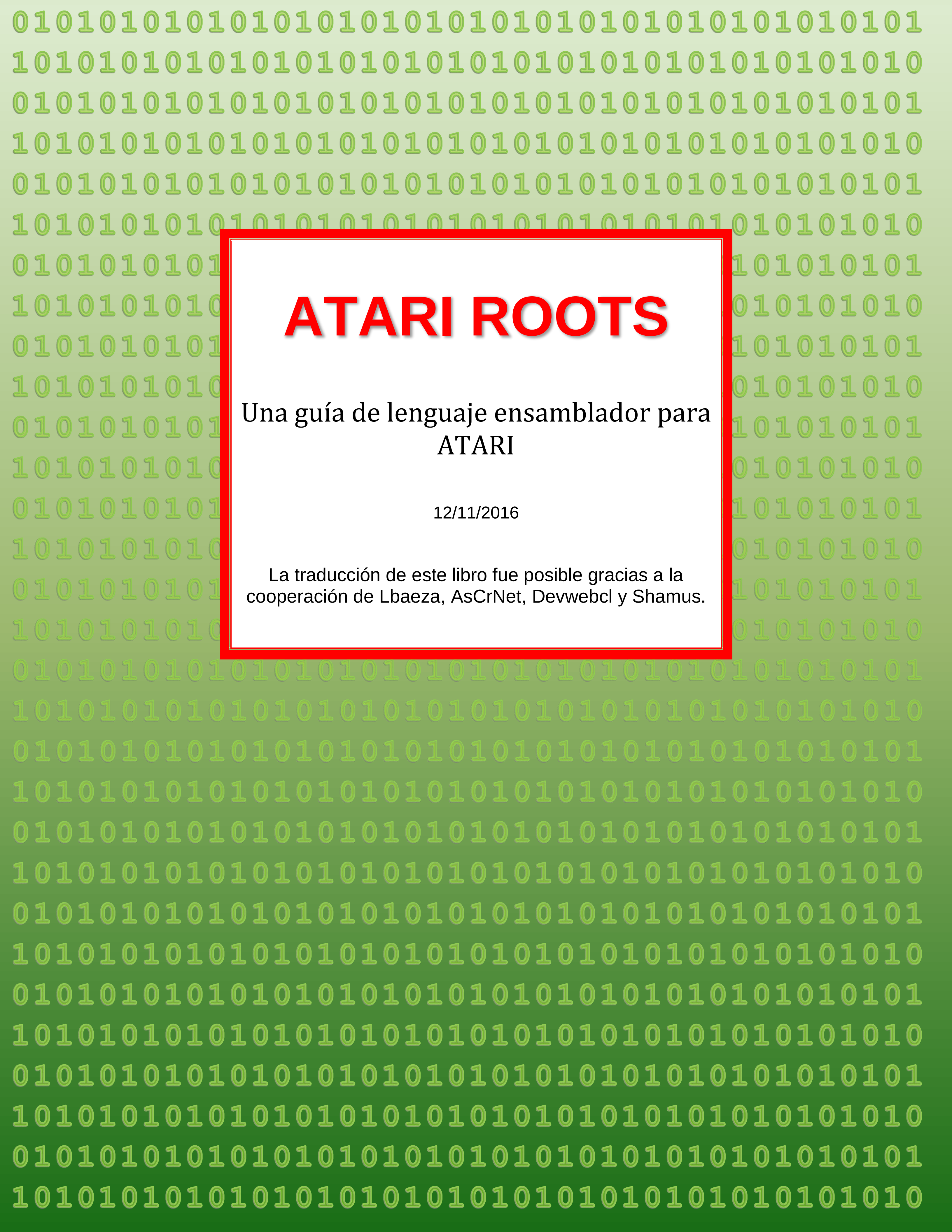


ATARI ROOTS

Una guía de lenguaje ensamblador para ATARI

12/11/2016

La traducción de este libro fue posible gracias a la cooperación de Lbaeza, AsCrNet, Devwebcl y Shamus.

Tabla de contenido

Prefacio

Se han escrito muchos libros de lenguaje ensamblador para el procesador 6502. Sin embargo, este libro es distinto a otros en varios aspectos. Hasta ahora, han existido dos grandes grupos de libros de lenguaje ensamblador para el procesador 6502. Están los libros genéricos que no se remiten a los computadores Atari, y por otro lado están los libros llenos de información acerca de los computadores Atari, pero que son tan técnicos que sólo pueden entenderlos los expertos. (También ha existido una escasez de libros para enseñar a programar en lenguaje ensamblador y que estén dirigidos a los usuarios que saben programar en el BASIC de ATARI).

Este libro fue escrito pensando en llenar este vacío. Está escrito en lenguaje humano, y no en el lenguaje de las computadoras. Este libro fue escrito pensando en los usuarios de computadores Atari, no en programadores profesionales (aunque ellos también pueden encontrar útil este libro). Cada tema que se toca ha sido ilustrado con a lo menos un programa simple que sea informativo y útil. Este libro también es único en otros aspectos. Por ejemplo, es la primera guía de lenguaje ensamblador que ha sido probada por los usuarios de la nueva serie XL de computadores Atari (aunque, cada programa de este libro también funcionará en los computadores Atari antiguos). Éste es el primer libro en referirse el uso del ensamblador MAC/65 de OSS, uno de los ensambladores más populares del mercado, y del cartucho Assembler Editor de Atari. Y no importa qué tipo de computador o de ensamblador posea, es muy probable que el libro que está leyendo sea la guía de lenguaje ensamblador más fácil de entender que alguna vez tendrá. En este libro encontrará todo lo que necesita saber para comenzar a ejecutar sus programas en un computador Atari. Lo mejor de todo es que este libro lo tendrá a usted escribiendo programas en lenguaje ensamblador antes de que lo sepa, y cuando termine, estará lo suficientemente bien encaminado como para llegar a ser un programador experto en lenguaje ensamblador.

Todo lo que necesita es este libro, un computador Atari (cualquier modelo), y las siguientes cosas:

- Un ensamblador y depurador (debugger) de lenguaje de máquina. Los programas de este libro funcionarán sin ningún cambio ya sea en el ensamblador MAC/65 de Optimized Systems Software (OSS) de San José, California, o en el cartucho Atari Assembler Editor fabricado por Atari. Si usted posee otro tipo de ensamblador, es muy probable que pueda utilizarlo sin mucha dificultad, ya que existe un conjunto de instrucciones estándar dentro del lenguaje ensamblador del 6502.
  Sin embargo, existen diferencias entre los ensambladores, tal como existen diferencias entre los dialectos usados por los distintos intérpretes de BASIC. Así que si usted usa un ensamblador distinto a los dos que fueron utilizados para escribir los programas de este libro, es probable que tenga que modificar la forma en que estos programas fueron escritos, ensamblados, depurados, cargados, guardados y ejecutados. No recomiendo hacer todo esto a menos que usted ya

sepa cómo programar en lenguaje ensamblador.

Cuando tenga en su poder este libro y un ensamblador, necesitará de las siguientes cosas para comenzar a programar en lenguaje ensamblador del Atari:

- Un cartucho de BASIC Atari y su Manual de Referencia.
- Una disquetera de 5 1/4 pulgadas compatible con los computadores Atari (Mejor aún, dos disqueteras).
- Una impresora Atari o compatible (De cualquier tipo -- de 40 columnas o de 80 columnas, térmica o de impacto, de matriz de puntos, no importa).

A medida que vaya aprendiendo más del lenguaje ensamblador, puede que necesite comprar más libros de computadores Atari y de programación en lenguaje ensamblador. Encontrará algunos de los mejores títulos que tratan estos temas en la bibliografía que se encuentra al final de este libro. ¡Buena suerte y feliz programación!

Mark Andrews
Nueva York, NY
Febrero de 1984

Introducción

Si su computador Atari no lo entiende, es probable que sea porque usted no habla su idioma. Juntos vamos a derribar esa barrera del lenguaje. Este libro le enseñará cómo escribir programas en lenguaje ensamblador -- el lenguaje de programación que corre más rápido y que maneja la memoria de la manera más eficiente. Este libro también le dará una buena idea acerca de cómo funciona el lenguaje de máquina, la lengua materna de su computador. Le permitirá crear programas imposibles de escribir en BASIC o en otros lenguajes menos avanzados. Y le probará que programar en lenguaje ensamblador no es tan difícil como usted pensaba.

Lo que le ofrecemos

Si usted sabe programar en BASIC - aunque sea sólo un poco - puede aprender a programar en lenguaje ensamblador; y una vez que haya aprendido, será capaz de hacer muchas otras cosas, tales como:

- Escribir programas que se ejecutarán entre 10 y 1000 veces más rápido que los programas escritos en BASIC.
- Desplegar por pantalla hasta 128 colores simultáneos.
- Diseñar sus propios lista de despliegue (display lists), mezclando texto y gráficos de la manera que usted desee.

También será capaz de:

- Crear su propio conjunto (set) de caracteres.
- Diseñar lista de despliegue (display lists) animados usando gráficos player-missile y las técnicas de animación de caracteres.
- Usar en sus programas desplazamiento (scrolling) horizontal y vertical, tanto fino como grueso.

Incluso descubrirá cómo:

- Crear efectos de sonido que son demasiado complejos para ser programados en BASIC.
- Usar los modos gráficos que no son soportados por BASIC.
- Escribir programas que arranquen desde el disco y se ejecuten automáticamente cuando usted encienda su computador.

En otras palabras, una vez que aprenda a programar en lenguaje ensamblador, será capaz de comenzar a escribir programas usando las mismas técnicas que usan los programadores profesionales de Atari. Muchas de estas técnicas son francamente imposibles de aprender sin un conocimiento acabado del lenguaje ensamblador. Por

último, y no por eso menos importante: A medida que vaya aprendiendo a programar en lenguaje ensamblador, irá descubriendo cómo es que funcionan los computadores. Y eso le permitirá ser un mejor programador en cualquier lenguaje.

Desmitificando el Lenguaje Ensamblador

Este libro ha sido cuidadosamente construido para terminar con el trabajo pesado que implica el aprendizaje del lenguaje ensamblador. Está lleno de programas de ejemplo y de rutinas. Incluso contiene una selección de programas interactivos, escritos en Atari BASIC, que fueron especialmente diseñados para ayudarle a aprender el lenguaje ensamblador.

El Capítulo 1 le presentará el lenguaje ensamblador y le explicará las diferencias entre este y otros lenguajes de programación.

En el capítulo 2 comenzará a descubrir los bits, los bytes y los números binarios, los “ladrillos” que usan los programadores para crear programas en lenguaje ensamblador. También encontrará algunos programas en BASIC que realizan automáticamente conversiones hexadecimales y binarias, lo que le ayudará a terminar con el misterio de los números hexadecimales y binarios.

En el capítulo 3 empezará a sondear los misterios del microprocesador 6502, el corazón (o más precisamente, el cerebro) de su computador Atari.

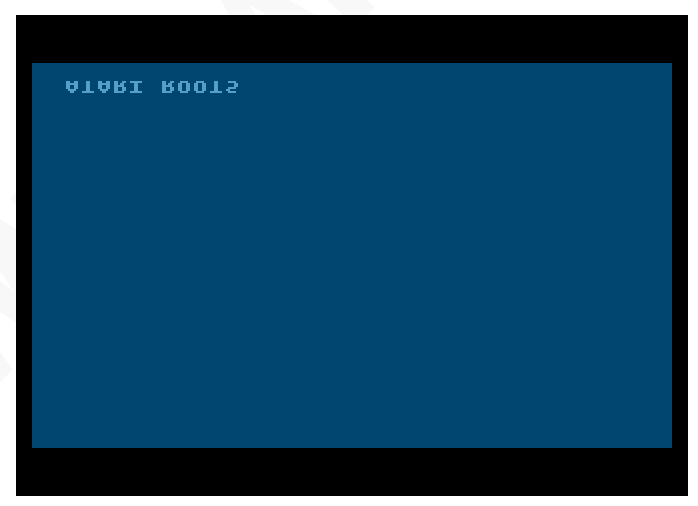
En el capítulo 4 comenzará a escribir programas en lenguaje ensamblador. Y al momento de terminar de leer este libro, estará bien encaminado para convertirse en un experto programador en lenguaje ensamblador.

Capítulo Uno
Introducción al lenguaje ensamblador

¡Comience a programar de inmediato en lenguaje de máquina! Encienda su computador Atari y digite el siguiente programa. A continuación, ejecútelo, y escriba algunas palabras. Verá algo muy interesante en la pantalla de su computador.

PROGRAMA DE EJEMPLO N° 1
"D:PATASARR.BAS"

```
10 REM ** "D:PATASARR.BAS" **
20 REM ** UN PROGRAMA EN LENGUAJE DE MAQUINA **
30 REM ** QUE PUEDES EJECUTAR **
40 REM ** PATAS ARRIBA **
50 REM
60 GRAPHICS 0 : PRINT
100 POKE 755, 4
110 OPEN #1,4,0,"K:"
120 GET #1,K
130 PRINT CHR$(K);
140 GOTO 120
```



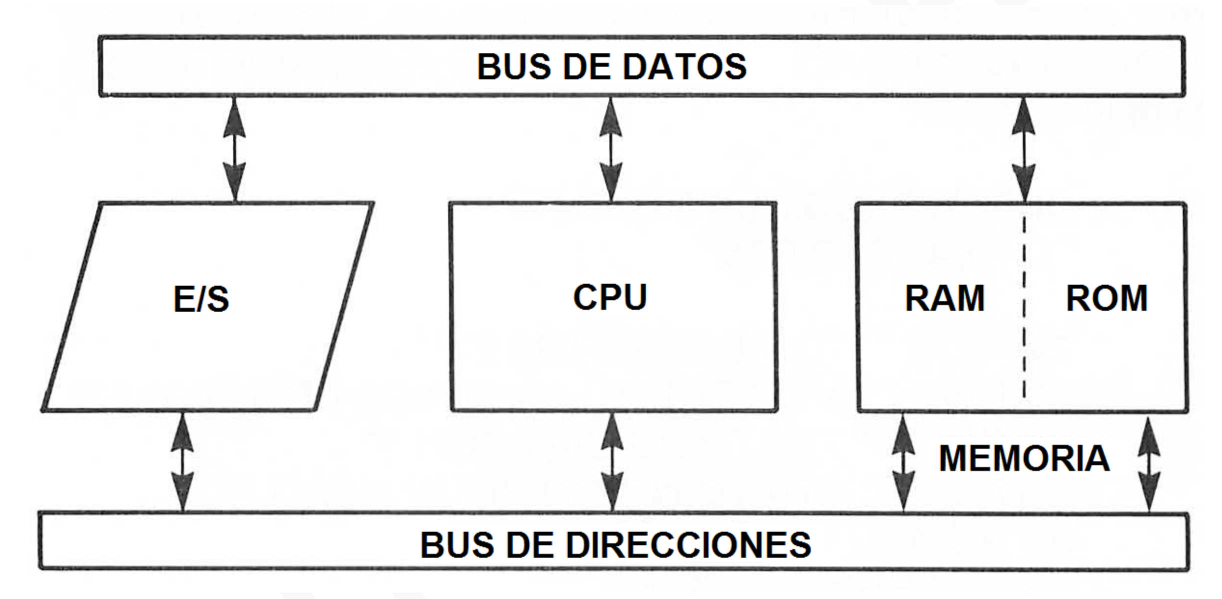
Captura de la pantalla

Por supuesto, este es un programa en BASIC. La línea 60 limpia la pantalla de su computador por medio del comando GRAPHICS 0. La línea 110 abre el teclado de su Atari como dispositivo de entrada. Luego, de las líneas 120 a la 140, hay un ciclo que imprime en la pantalla los caracteres que se han digitado. Pero la línea más importante de este programa, es la línea 100. El ingrediente activo de la línea 100, la instrucción POKE 755,4, es en realidad una instrucción de lenguaje de máquina. De hecho, todos los

comandos POKE en BASIC son instrucciones en lenguaje de máquina. Cuando se utiliza un comando POKE en BASIC, lo que usted está haciendo es almacenar un número en una posición de memoria específica de su computador. Y cuando se almacena un número en una posición de memoria específica de su computador, lo que está haciendo es usar lenguaje de máquina.

Bajo el capó de su Atari

Cada computador tiene tres partes principales: una unidad de procesamiento central (Central Processing Unit - CPU), la memoria (por lo general dividida en dos bloques llamados memoria de acceso aleatorio (Random Access Memory - RAM) y memoria de sólo lectura (Read Only Memory - ROM), y los dispositivos de entrada / salida - E/S (Input/Output - I/O).



Principales partes del computador hogareño Atari

El dispositivo de entrada principal de su Atari es el teclado. Su dispositivo de salida principal es su monitor de video. Otros dispositivos de E/S a los que se puede conectar un computador Atari son los módems telefónicos, tableros gráficos, grabadores de cassettes, y unidades de disco. En un microcomputador, todas las funciones de una unidad central de procesamiento están contenidas en una Unidad de Microprocesador o MPU (MicroProcessor Unit).La MPU de su computador Atari, así como su Unidad Central de Proceso (Central Processing Unit - CPU), es un circuito a Gran Escala de Integración (Large Scale Integration - LSI) llamado microprocesador 6502.

La Familia 6502

El microprocesador 6502, centro de mando de su equipo, fue desarrollado por MOS Technology, Inc. Varias compañías ahora poseen la licencia para la fabricación de chips 6502, y algunos fabricantes de computadores lo usan en sus máquinas. El chip 6502 y

varios modelos actualizados, tales como el 6502A y el 6510, no sólo son utilizados en los computadores Atari, sino también en los computadores personales fabricados por Apple, Commodore, y Ohio Scientific.

Eso significa, por supuesto, que el lenguaje ensamblador 6502 también se puede utilizar para programar diferentes computadores personales - incluyendo el Apple II, Apple II +, Apple //e y Apple ///, todos los computadores de Ohio Scientific, el computador Commodore PET, y el Commodore 64.

Y eso no es todo; los principios utilizados en el lenguaje de programación ensamblador de Atari son universales; son los mismos principios que usan los programadores en lenguaje ensamblador, sin importar para qué tipo de computador están escribiendo sus programas.

Una vez que aprenda el lenguaje ensamblador 6502, le será fácil aprender a programar otro tipo de procesador, como el chip Z-80 usado en computadores Radio Shack y equipos basados en CP/M, e incluso los chips poderosos más nuevos que se utilizan en micro-computadores de 16-bits, como el IBM-PC.

Las Fuentes de ROM

Su equipo tiene dos tipos de memoria: Memoria de Acceso Aleatorio (RAM) y Memoria de Sólo Lectura (ROM). ROM es la memoria a largo plazo de su Atari. Fue instalada en su equipo en la fábrica, y es tan permanente como su teclado. La ROM de su equipo está permanentemente grabada en un grupo determinado de chips, así que nunca se borra, incluso cuando se apaga el suministro de corriente. Para los dueños de la mayoría de los equipos hogareños, esto es algo bueno.

Sin su ROM, su Atari no sería un Atari. De hecho, no sería mucho más que un carísimo pisapapeles de alta tecnología. El mayor bloque de memoria en la ROM es el bloque que contiene el Sistema Operativo (Operating System - OS) de su computador. El sistema operativo de su Atari es lo que le permite hacer todas esas cosas maravillosas que se supone que los Ataris tienen que hacer, como permitir el ingreso de datos desde el teclado, desplegar caracteres en la pantalla, etc.

También la ROM es lo que permite a su equipo comunicarse con dispositivos periféricos como discos duros, grabadoras de cassettes, y módems telefónicos. Si usted es dueño de un computador Atari de la serie XL, el paquete ROM de su unidad también contiene una serie de características adicionales, como un sistema de auto-diagnóstico, un conjunto de caracteres de idiomas extranjeros, y lenguaje BASIC incorporados.

La RAM es volátil

La ROM, como puede imaginar, no se construyó en un día. El paquete ROM de su Atari es el resultado de un montón de trabajo en lenguaje ensamblador hecho por una gran

cantidad de programadores. El contenido de la RAM, por otra parte, puede ser escrito por cualquier persona - incluso usted. La memoria RAM es la memoria principal del computador. Tiene muchas más celdas de memoria que la ROM, pero la RAM, a diferencia de la ROM, es volátil.
El problema con la RAM es que se puede borrar, o, como diría un ingeniero informático, es volátil. Cuando usted enciende su computador, el bloque de memoria en su interior que está reservado para la RAM está tan vacío como una hoja de papel en blanco.

Y cuando usted apaga su equipo, desaparece cualquier cosa que usted pueda tener en la memoria RAM. Es por eso que la mayoría de los programas de computadores tienen que ser cargados en la memoria RAM desde de los dispositivos de almacenamiento masivo, tales como grabadoras de cassettes y unidades de disco. Después de haber escrito un programa, usted tiene que guardarlo en algún lugar para que no se borre cuando se desconecte la corriente y se borre la memoria RAM.

La memoria RAM de su computador, o memoria principal, se puede imaginar como una red enorme formada por miles de compartimentos, o celdas, algo así como filas sobre filas de casillas de correos a lo largo de una pared. Cada celda de esta gran matriz de memoria se llama posición de memoria, y cada posición de memoria, como cada casilla en una oficina de correos, tiene una dirección de memoria individual y única.

La analogía entre los computadores y las casillas de correos no termina allí. Un programa, como un trabajador postal experto que pone cartas en las casillas de la oficina de correos, puede llegar rápidamente a cualquier posición de memoria. En otras palabras, se puede acceder a cualquier posición en su memoria al azar. Y es por eso que a la memoria de un computador que es direccionable por el usuario se le conoce como memoria de acceso aleatorio.

Las "Cartas" son Números

Sin embargo, nuestra analogía de la oficina de correos no es absolutamente perfecta. Una casilla de la oficina de correos puede llenarse con muchas cartas, pero cada posición de memoria de un computador sólo puede tener un solo número. Y ese número puede representar sólo una de tres cosas:

- El número almacenado en sí;
- Un código que representa un carácter, o
- Una instrucción de lenguaje de máquina.

¿Qué viene a continuación?

Cuando un computador se dirige a una posición de memoria y encuentra un número, se le debe decir qué hacer con el número que encontró. Si el número equivale a un simple número, entonces se le debe decir al computador por qué el número está ahí. Si el

número es un código que representa un carácter, a continuación, se le debe decir al computador cómo se va a utilizar ese carácter. Y si el número debe ser interpretado como una instrucción de lenguaje de máquina, también se le debe decir al computador que se trata de una instrucción de lenguaje de máquina.

Las instrucciones son programas

Las instrucciones que se le dan a las computadores para que puedan encontrar e interpretar los números almacenados en la memoria se denominan programas de computador. La gente que escribe programas son, por supuesto, programadores. Los lenguajes en que los programas están escritos se llaman lenguajes de programación. De todos los lenguajes de programación, el ensamblador es el más global.

Ejecutando un programa en Lenguaje de Máquina

Cuando su computador va a ejecutar un programa, lo primero que se le tiene que decir es en qué lugar de su memoria está almacenado el programa. Una vez que tenga dicha información, podrá ir a la dirección de memoria donde comienza el programa y echar un vistazo a lo que hay allí.

Si el equipo encuentra una instrucción que está programado para entender, entonces ejecutará dicha instrucción. El computador pasará a la siguiente dirección en su memoria. Después de seguir las instrucciones que encuentre allí, se pasará a la siguiente dirección, y así sucesivamente.

El equipo repetirá este proceso de ejecutar una instrucción y pasar a la siguiente, hasta que llega al final de cualquier programa que se ha almacenado en su memoria. Entonces, a menos que encuentre una instrucción para regresar a una dirección en el programa o para saltar a una nueva dirección, simplemente se queda, esperando pacientemente recibir otra instrucción.

Lenguajes de Computadores

Como ustedes saben, los programas pueden ser escritos en decenas de lenguajes de programación tales como BASIC, COBOL, Pascal, LOGO, etc. Lenguajes como éstos son llamados lenguajes de alto nivel, no porque sean especialmente esotéricos o profundos, sino porque están escritos en un nivel demasiado alto para que un computador lo pueda entender.

Un computador puede entender sólo un lenguaje, el lenguaje de máquina, el cual está escrito totalmente en números. Así que antes de que un computador pueda ejecutar un programa escrito en un lenguaje de alto nivel, el programa de alguna manera debe ser traducido a lenguaje de máquina.

Los programas escritos en lenguajes de alto nivel usualmente son traducidos a lenguaje de máquina por medio del uso de paquetes de software llamados intérpretes y compiladores. Un intérprete es un software que puede convertir un programa a lenguaje de máquina, a medida que éste está siendo escrito.

El intérprete BASIC de su Atari es un intérprete de lenguaje de alto nivel. Los intérpretes se usan para convertir programas escritos en otros lenguajes de alto nivel, como LOGO y Pilot, en programas de lenguaje de máquina. Un compilador es un paquete de software diseñado para convertir los lenguajes de alto nivel en lenguaje de máquina después de que éstos se hayan escrito. COBOL, Pascal y la mayoría de los lenguajes de alto nivel se suelen traducir a lenguaje de máquina con la ayuda de los compiladores.

Ensambladores de Lenguaje de Máquina

Los intérpretes y compiladores no se utilizan en la escritura de programas en lenguaje ensamblador. Los programas de lenguaje de máquina casi siempre son escritos con la ayuda de paquetes de software llamados ensambladores.

Existen otros ensambladores para el computador Atari, tales como el paquete avanzado Macro Assembler y Text Editor de Atari. Un ensamblador no funciona como un intérprete, o como un compilador. Esto es porque el lenguaje ensamblador no es un lenguaje de alto nivel. En efecto, se podría decir que el lenguaje ensamblador no es realmente un lenguaje de programación.

En realidad, el lenguaje ensamblador no es más que un sistema de notación utilizado para escribir programas en lenguaje de máquina utilizando símbolos alfabéticos que los programadores humanos pueden entender.
Lo que estamos tratando de hacer entender aquí es el hecho de que el lenguaje ensamblador es totalmente diferente de cualquier otro lenguaje de programación. Cuando un programa en lenguaje de alto nivel es traducido a lenguaje de máquina por un intérprete o compilador, una sola instrucción en el lenguaje original fácilmente puede equivaler a decenas - a veces incluso centenares - de instrucciones de lenguaje de máquina. Sin embargo, cuando se escribe un programa en lenguaje ensamblador, cada instrucción que es usada en el lenguaje ensamblador equivale a una sola instrucción de lenguaje de máquina con exactamente el mismo significado. En otras palabras, hay una relación exacta de uno a uno entre la instrucción en lenguaje ensamblador y las instrucciones en lenguaje máquina. Debido a esta correspondencia única, los ensambladores de lenguaje de máquina tienen un trabajo mucho más fácil que el que realizan los intérpretes y compiladores.

Dado que los programas en lenguaje ensamblador (a menudo llamado el código fuente) se pueden convertir directamente en programas de lenguaje de máquina (a menudo conocido como código objeto), un ensamblador puede simplemente pasar como un rayo traduciendo listados de código fuente en código objeto, sin tener que luchar con todas las

tortuosas contorsiones de traducción que los intérpretes tienen que enfrentar cada vez que llevan a cabo sus tareas.

Los ensambladores también tienen una ventaja sobre los compiladores. Los programas que producen tienden a ser más sencillos y menos repetitivos. Los programas ensamblados son más eficientes con la memoria y corren más rápido que los programas interpretados o compilados.

La difícil situación del Programador

Desafortunadamente, se debe pagar un precio por toda esta eficiencia y velocidad; y el individuo que paga este precio es, tristemente, el programador en lenguaje ensamblador. Irónicamente, a pesar de que los programas en lenguaje ensamblador funcionarán mucho más rápido que los programas escritos en lenguaje de alto nivel, estos requieren de muchas más instrucciones y toman mucho más tiempo en ser escritos.

Una estimación ampliamente citada es que a un programador experto le toma cerca de diez veces más tiempo el escribir un programa en lenguaje ensamblador que lo que le tomaría escribir el mismo programa en un lenguaje de alto nivel como BASIC, COBOL, o Pascal. Por otra parte, los programas en lenguaje ensamblador se ejecutan 10 a 1000 veces más rápido que los programas en BASIC, y pueden hacer cosas que los programas en BASIC no pueden hacer a ninguna velocidad. Así que si quiere convertirse en un programador experto, realmente no tiene más remedio que aprender el lenguaje ensamblador.

Cómo funciona el Lenguaje de Máquina

El lenguaje de máquina, como cualquier otro lenguaje de computadores, se compone de instrucciones. Sin embargo, como hemos señalado, cada instrucción que se usa en lenguaje de máquina es un número. Los números que entienden los computadores no son del tipo que estamos acostumbrados a usar. Los computadores piensan en números binarios - números que no son más que secuencias de unos y ceros. Aquí tenemos, por ejemplo, una parte de un programa de computador escrito en números binarios (el tipo de números que un computador entiende):

```
00011000
11011000
10101001
00000010
01101001
00000010
10000101
11001011
01100000
```

No hace falta mucha imaginación para ver que usted estaría en un problema si tiene que escribir programas extensos, en el estilo binario del lenguaje de máquina que normalmente contienen miles de instrucciones. Con un ensamblador, sin embargo, la tarea de escribir un programa en lenguaje de máquina es considerablemente más fácil. Aquí, por ejemplo, está el programa anterior, tal como aparecería si usted lo hubiera escrito en lenguaje ensamblador:

```
CLC
CLD
LDA
#02
ADC
#02
STA
$CB
RTS
```

Puede no entender todo eso todavía, pero tendrá que admitir que por lo menos se ve más comprensible. Lo que hace el programa, por cierto, es sumar 2 y 2. Luego se almacena el resultado del cálculo en una ubicación de memoria determinada de su computador - en concreto, la dirección de memoria 203. Más adelante volveremos a este programa y le daremos una mirada más en detalle. Entonces tendrá la oportunidad de ver exactamente cómo funciona. Antes, eso sí, vamos a enfocarnos en los ensambladores y el lenguaje ensamblador.

Comparación entre Lenguaje Ensamblador y BASIC

Los programas en lenguaje ensamblador se escriben usando instrucciones de tres letras llamadas mnemotécnicos. Algunos mnemotécnicos son similares a instrucciones BASIC. Una instrucción en lenguaje ensamblador que es muy similar a una instrucción BASIC es RTS, la última instrucción en la rutina de ejemplo que acabamos de revisar. RTS (0110 0000 escrito en lenguaje de máquina) significa "regrese de la subrutina" ("ReTurn from Subroutine.").

Se utiliza de manera similar a la instrucción RETURN de BASIC. También hay un mnemotécnico del lenguaje ensamblador que es similar a la instrucción GOSUB de BASIC. Se escribe JSR, y significa salte a la subrutina ("Jump to SuBroutine."). Su equivalente en código binario de lenguaje de máquina es 0010 000.

Sin embargo, no todas las instrucciones en lenguaje ensamblador tienen tal parecido con las instrucciones del BASIC. Una instrucción en lenguaje ensamblador nunca le ordena al computador ejecutar algo tan complejo como dibujar una línea o imprimir una letra en la pantalla, por ejemplo.

En cambio, la mayoría de los mnemónicos del lenguaje ensamblador ordenan a los computadores ejecutar tareas muy elementales como la suma de dos números, comparar dos datos, o (como hemos visto) saltar a una subrutina. Es por eso que a veces se requiere un gran número de instrucciones de lenguaje ensamblador para igualar a una o dos instrucciones en un lenguaje de alto nivel.

Código Fuente y Código Objeto

Cuando escribe un programa en lenguaje ensamblador, el listado que usted produce se llama código fuente, ya que es la fuente de la que un programa de lenguaje de máquina será producido. Una vez que ha escrito el código fuente de un programa en lenguaje ensamblador, puede pasarlo por un ensamblador. El ensamblador lo convertirá en un código objeto, que es sólo otro nombre para un programa en lenguaje de máquina producido por un ensamblador.

La Velocidad y Eficiencia del Lenguaje de Máquina

Dado que las instrucciones en lenguaje ensamblador son tan específicas (incluso podríamos decir primitivas), obviamente toma muchas de ellas el hacer un programa completo, y muchas, muchas más instrucciones de lo que tomaría escribir el mismo programa en un lenguaje de alto nivel.

Irónicamente, los programas en lenguaje de máquina aún ocupan menos espacio en memoria que los programas escritos en lenguajes de alto nivel. Esto es porque cuando un programa escrito en un lenguaje de alto nivel es interpretado o compilado en lenguaje de máquina, grandes bloques de código de máquina deben ser repetidos cada vez que se utilizan. Pero en un buen programa escrito en lenguaje ensamblador, una rutina que se utiliza una y otra vez puede ser escrita sólo una vez, y luego será utilizada cuantas veces como sea necesario por medio de JSR, RTS, y otros comandos similares. Muchos otros tipos de técnicas pueden utilizarse para conservar la memoria en los programas escritos en lenguaje ensamblador.

Capítulo Dos
Bits, Bytes y Binario

Un uno o un cero pueden no significar mucho para usted, pero para un computador significan mucho. Los números binarios, como usted ya sabe, son números compuestos únicamente de unos y ceros. Y son el único tipo de números que un computador puede entender. Un computador, o en todo caso, un computador digital (su Atari lo es), tiene grandes dificultades para entender conceptos poco claros. Para un computador, un interruptor está encendido o está apagado. Una señal eléctrica está presente o no lo está. Cada cosa en la pequeña mente de un microcomputador es blanco o negro, positivo o negativo, encendido o apagado.

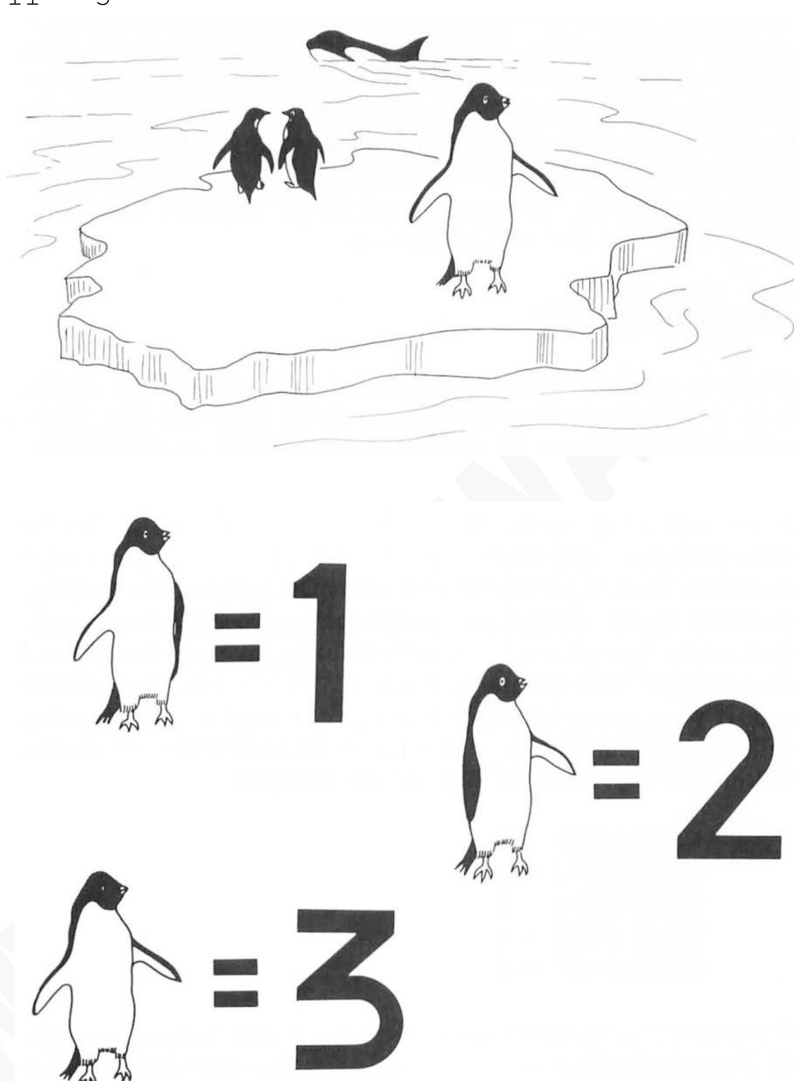
Llámelo como quiera. No importa. Ya sea masculino y femenino, Shiva y Shakti, o el yin y el yang. A veces los matemáticos la llaman el álgebra de Boole. Los diseñadores de computadores a veces la llaman lógica de dos estados. Y los programadores a menudo se refieren a ella como el sistema binario. En el sistema binario, el dígito 1 simboliza lo positivo, una corriente que fluye, por ejemplo, o un interruptor que está encendido. El dígito 0 representa lo negativo, una corriente que no fluye, o un interruptor que está apagado. Pero no hay dígito para el número 2. La única manera de representar el número 2 en binario es tomar un 1, moverlo un espacio a la izquierda, y poner a continuación un 0, de esta manera: 10. Eso es lo correcto. En notación binaria, "10" significa 2, no 10, "11" significa 3, "100" es 4, "101" es 5, y "110" es 6, y así sucesivamente.

Matemáticas Pingüinas

Si los números binarios lo frustran, un curso de Matemáticas Pingüinas el ayudara. Imagínese que usted es un pingüino, viviendo en un campo de hielo. Los pingüinos no tienen 10 dedos en cada mano, como los tiene la gente. En cambio, tienen dos aletas. Así que si usted fuera un pingüino, y contara sus aletas tal como algunas personas cuentan sus dedos, usted sería capaz de contar sólo hasta 2. Sin embargo, si usted fuera un pingüino muy brillante, es posible que un día descifre cómo usar sus aletas para contar más allá de 2. Supongamos, por ejemplo, que se decidió a definir que la aleta derecha levantada es igual a 1, y que aleta izquierda levantada es igual a 2, entonces usted puede decir que las dos aletas levantadas equivalen a 3. Ahora supongamos que usted fuera un pingüino extraordinariamente brillante, y diseñó un sistema de notación para expresar por escrito lo que ha descubierto. Usted podría utilizar un 0 para representar a ninguna aleta levantada, y un 1 para representar una aleta levantada. Y entonces usted podría escribir estas ecuaciones en el hielo:

Números Pingüinos

```
00 = 0
01 = 1
10 = 2
11 = 3
```



Estos, por supuesto, son números binarios. Y lo que muestran, en Matemáticas Pingüinas, es que se pueden expresar cuatro valores - del 0 al 3 - como números binarios de 2-bits (dos dígitos). Ahora vamos a suponer que usted, como pingüino, quiere aprender a contar más allá del 3. Imaginemos que usted miró sus pies y se dio cuenta que tenía dos aletas más. ¡Voilá, números más grandes! Si usted se sentara en su campo de hielo de modo que usted pudiera levantar ambos brazos y ambas piernas al mismo tiempo, usted puede contar de la siguiente manera:

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
```

... y así sucesivamente.

Si sigue contando así finalmente descubriría que puede expresar 16 valores, del 0 al 15, usando números de 4 bits. Sólo queda una lección más de Matemáticas Pingüinas. Imagine que usted, como buen pingüino, se ha casado con una pingüina. Y, utilizando su habilidad con los números binarios, determinó que entre usted y su pareja hacen un total de ocho aletas. Si su pareja decide cooperar con usted y contar también con sus aletas, los dos ahora podrían sentarse en el hielo y comenzar un paseo en témpano con números que se verían de la siguiente manera:

```
0000 0001 = 1
0000 0010 = 2
0000 0011 = 3
0000 0100 = 4
0000 0101 = 5
```

Si usted y su pareja siguieran contando de esta manera, mediante Matemáticas Pingüinas de 8 bits (aletas), finalmente descubriría que con ocho aletas se puede contar desde 0 hasta 255, lo que da un total de 256 valores. Esto completa nuestro breve curso en Matemáticas Pingüinas. Lo que nos ha enseñado es que es posible expresar 256 valores, del 0 a 255, usando números binarios de 8 bits.

Bits, Bytes y Nibble

Como se ha señalado unos párrafos atrás, cuando unos y ceros se usan para expresar números binarios, se les llaman bits. Un grupo de ocho bits se llama byte. Y un grupo de cuatro bits se llama nibble (a veces se escribe "nybble"). Y un grupo de 16 bits se le denomina palabra (word). Ahora vamos a echar otro vistazo a una serie de bytes de 8 bits. Obsérvelos de cerca, y verá que cada número binario que termina en cero es dos veces mayor que el número anterior, o en otras palabras, es el cuadrado del número anterior:

```
0000 0001 = 1
0000 0010 = 2
0000 0100 = 4
0000 1000 = 8
0001 0000 = 16
0010 0000 = 32
0100 0000 = 64
1000 0000 = 128
```

Acá hay dos números binarios más que los programadores de lenguaje ensamblador a menudo encuentran útil memorizarlos:

```
1111 1111 = 255
1111 1111 1111 1111 = 65,535
```

El número 255 es digno de mención porque es el número de 8 bits más grande. El número 65.535 es el número de 16 bits más grande. Ahora la trama se complica. Como ya hemos mencionado, los computadores Atari se llaman computadores de 8 bits porque están construidos alrededor de un microprocesador de 8 bits, un procesador de computadores que maneja números binarios de hasta sólo ocho posiciones de largo. Debido a esta limitación, el Atari no puede realizar cálculos con números mayores a 255, de hecho... ¡Ni siquiera puede realizar un cálculo cuyo resultado sea mayor a 255!

Obviamente, esta limitación de los 8-bits impone severas restricciones sobre la capacidad de los computadores Atari para realizar cálculos con números grandes. En efecto, la Unidad Aritmética Lógica (Arithmetic Logic Unit - ALU) del 6502 es como una calculadora que no puede manejar un número mayor a 255. Hay maneras de eludir esta limitación, por supuesto, pero no es fácil. Para trabajar con números de más de 255, un computador de 8 bits tiene que realizar una serie más compleja de las operaciones. Si un número es mayor que 255, un computador de 8 bits tiene que dividirlo en trozos 8-bits, y realizar cada cálculo requerido en cada número de 8 bits. Luego, el computador tiene que juntar todos estos números de 8 bits.

Si el resultado de un cálculo tiene más de ocho bits de largo, las cosas se complican aún más. Esto porque cada posición de memoria en un computador de 8 bits, cada celda en su memoria de acceso aleatorio (RAM), así como su memoria de sólo lectura (ROM), es un registro de memoria de 8 bits. Así que si desea almacenar un número mayor que 255 en la memoria de un computador de 8-bits, usted tiene que dividirlo en dos o más números de 8 bits, y después almacenar cada uno de estos números en un lugar de memoria independiente. Y después, si alguna vez desea utilizar el número original de nuevo, tiene que juntar cada uno de los trozos de 8 bits en que fue dividido anteriormente.

Computadores de 8-bits versus 16-bits

Ahora que sabe todo eso, podrá entender por qué los computadores de 16 bits, como los computadores personales IBM y sus compatibles, pueden correr más rápido que los computadores de 8 bits. Un equipo de 16 bits puede manejar números binarios de 16 bits de largo, sin hacer operaciones matemáticas cortando y pegando números, y por lo tanto pueden procesar números que llegan hasta el 65.535 en trozos individuales, una palabra de 16 bits a la vez. Así que los computadores de 16 bits son considerablemente más rápidos que los de 8 bits, por lo menos cuando están operando con números grandes. Los computadores de 16 bits también tienen otra ventaja sobre los de 8 bits. Pueden estar al tanto de mucha más información simultánea que los computadores de 8 bits. Y por lo tanto, puede ser equipado con memorias mucho más grandes.

Mapa de Memoria de su Computador

La memoria de un computador puede ser visualizada como una cuadrícula grande que contiene miles de casillas, o posiciones de memoria. En un computador Atari, cada uno de estas posiciones puede contener un número de 8 bits. Anteriormente, se presentó una analogía entre la memoria de un computador y las filas sobre filas de casillas de correos. Ahora podemos ampliar esa analogía un poco más.

Como hemos mencionado, cada posición de memoria en su computador tiene una dirección de memoria individual y única. Y cada una de estas direcciones está en realidad compuesta de dos números índice. El primero de estos números se podría llamar el eje X de la posición de la dirección en la matriz de memoria de su computador. El segundo número se podría llamar el eje Y de la posición. Al utilizar este sistema de coordenadas X, Y para localizar las direcciones en su memoria, el computador Atari puede estar al tanto de las direcciones que tienen hasta 16 bits de largo, aunque sólo se trate de un computador de 8-bits. Todo lo que tiene que hacer es almacenar el eje X en un registro de 8 bits y el eje Y en otro registro de 8 bits. Así que cuando usted quiera que su Atari busque un trozo de información de su memoria, todo lo que tienes que hacer es indicar cuál es el eje X y el eje Y de la dirección donde están los datos. Así el computador puede encontrar inmediatamente los datos que usted está buscando.

La Libreta de Direcciones de su Computador

Pero aun así hay un límite en el número de posiciones de direcciones que un computador de 8 bits puede estar al tanto. Dado que 255 es el número más grande que un registro de 8 bits puede almacenar, el eje X de una dirección puede ir sólo de 0 a 255, o sea un total de 256 números. El mismo límite se aplica al eje Y. Así que a menos que se utilicen elaborados trucos de expansión de memoria, la capacidad máxima de un computador de 8 bits es 256 veces 256, o sea 65.536, es decir, 64K de memoria. La razón de este número, por cierto, es que 64 K equivale a un número binario, no un a número decimal. 64K no es igual a 64 multiplicado por 1.000, en cambio equivale al producto de 64 y 1024.

Los dos números cuando están escritos en notación decimal pareciera que fueron obtenidos como por arte de magia, pero en binario ambos son números redondos: 0100 000 y 0100 000 000, respectivamente.

La Diferencia Que Hacen los 8 Bits

Si un computador de 16-bits está al tanto de las direcciones en su memoria de la misma manera que un computador de 8-bits, utilizando el eje X y el eje Y de cada posición como puntos de referencia, entonces un equipo de 16 bits puede direccionar más de 4 millones de posiciones de memoria (65.536 celdas por 65.536 celdas). Sin embargo, los computadores de 16-bits no suelen estar al tanto de la dirección en memoria de esa manera. En el IBM-PC, por ejemplo, a cada posición de memoria se le asigna lo que equivale a una dirección de 20 bits. Así, un IBM-PC puede direccionar 1.048.576 posiciones; no son exactamente los más de 4 millones de posiciones que podría abordar con el sistema de la matriz X, Y, pero aún suficientes posiciones para almacenar más de un millón de bytes (un megabyte) de memoria.

Ahora puede entender de qué se trata todo este alboroto acerca de los computadores de 16 bits. Pueden direccionar más memoria que los computadores de 8 bits, y también pueden procesar información más rápido. Además, son más fáciles de programar en lenguaje ensamblador que los computadores de 8-bits, ya que pueden manipular de manera natural fragmentos de datos de 16 bits de longitud. Dado que los computadores Atari son de 8 bits, nada de esto aplicable a los computadores de 16 bits será de mucha ayuda en su búsqueda de conocimientos sobre el lenguaje ensamblador de Atari. Afortunadamente, sin embargo, un conocimiento del lenguaje ensamblador de Atari le ayudará mucho si alguna vez decide estudiar lenguaje ensamblador de 16-bits. Si aprende a jugar con bits lo suficientemente bien como para convertirse en un buen programador de lenguaje ensamblador de Atari, probablemente será más rápido aprender el lenguaje ensamblador de 16-bits, ya que éste no requiere acoplar números de 8 bits.

El Sistema Numérico Hexadecimal

¿Cuál es la suma de D más F? Bueno, es 1C, si usted está trabajando en números hexadecimales.

Los números hexadecimales, como puede saberlo si usted ha trabajado mucho programando, son extrañas combinaciones de letras y números que se utilizan a menudo en programas en lenguaje ensamblador. El sistema de notación hexadecimal utiliza no sólo los dígitos del 0 al 9, sino también las letras A a la F. Así que combinaciones de letras y números de tan extraño aspecto como FC3B, 4A5D e incluso ABCD son números perfectamente válidos en el sistema hexadecimal. Los números hexadecimales son utilizados a menudo por los programadores en lenguaje ensamblador, ya que están muy relacionados con los números binarios. A continuación echaremos un vistazo a esta estrecha relación.

Matemáticas con 16 Dedos

¿Recuerda cómo usamos las Matemáticas Pingüinas para explicar el concepto de números binarios? Bueno, si puede imaginar que vive en una sociedad donde cada individuo tiene 16 dedos en vez de 10 como nosotros, o dos aletas como un pingüino, entonces será capaz de comprender el concepto de números hexadecimales con bastante facilidad. Los números binarios, como hemos señalado, tienen base 2.

Los números decimales, del tipo que estamos acostumbrados, tienen base 10. Y los números hexadecimales tienen base 16. Los números hexadecimales se utilizan en la programación en lenguaje ensamblador, porque son muy similares a los números binarios, lo cual, como hemos señalado en el Capítulo 1, son el tipo de números que entienden los computadores. A primera vista, puede ser difícil ver qué tienen en común los números binarios y los hexadecimales. Pero vera claramente cómo los números binarios y hexadecimales se relacionan entre sí, mirando este gráfico:

Decimal	Hexadecimal	Binario
1	1	0000 0001
2	2	0000 0010
3	3	0000 0011
4	4	0000 0100
5	5	0000 0101
6	6	0000 0110
7	7	0000 0111
8	8	0000 1000
9	9	0000 1001
10	A	0000 1010
11	B	0000 1011
12	C	0000 1100
13	D	0000 1101
14	E	0000 1110
15	F	0000 1111
16	10	0001 0000

Como puede ver en esta lista, el número decimal 16 se escribe "10" en hexadecimal y "0001 0000" en binario, y es por tanto un número redondo en ambos sistemas. Y el dígito hexadecimal F, que viene justo antes del 10 hexadecimal (o 16 decimal), se escribe en binario 00001111. A medida que se familiarice con los sistemas binario y hexadecimal, comenzará a notar muchas otras similitudes entre ellos. Por ejemplo, el número decimal 255 (el mayor número de 8 bits) es 1111 1111 en binario y FF en hexadecimal. El número decimal 65.535 (la dirección de memoria más alta en un equipo de 64K) se escribe FFFF en hexadecimal y 1111 1111 1111 1111 en binario. Y así sucesivamente. El punto de todo

esto es que es mucho más fácil de convertir entre números binarios y hexadecimales, que convertir entre números binarios y decimales.

```
1011  1000  Binario
B     8     Hexadecimal
184         Decimal

0010  1110  Binario
2     E     Hexadecimal
46          Decimal

1111  1100  Binario
F     C     Hexadecimal
252         Decimal

0001  1100  Binario
1     C     Hexadecimal
28          Decimal
```

Como puede ver, un nibble (cuatro bits) en notación binaria siempre equivale a un dígito en notación hexadecimal. Pero no hay ninguna relación clara entre el largo de un número binario y el largo del mismo número escrito en notación decimal. Este mismo principio puede extenderse a conversiones de binario, hexadecimal y decimal de números de 16 bits. Por ejemplo:

```
1111  1100  0001  1100  Binario
F     C     1     C     Hexadecimal
64540                   Decimal
```

Algunos programas ilustrativos

Ahora vamos a mirar algunos programas básicos que llevan a cabo operaciones con números binarios, decimales y hexadecimales. Dado que todos los computadores Atari son computadores de 8 bits (al momento de la escritura de este libro, de todos modos), la única forma de almacenar un número de 16 bits en un Atari es ponerlo en dos registros de la memoria. Y los computadores basados en el 6502 usan la rara convención (para algunas personas) de almacenar números de 16 bits de la siguiente manera: primero el byte más bajo (menos significativo) y luego el byte más alto (más significativo). Por ejemplo, si el número hexadecimal FC1C va a ser almacenado en las direcciones de memoria hexadecimal 0600 y 0601, FC (el byte más significativo) se almacenará en la dirección 0601, y 1C (el byte menos significativo) se almacenará en la dirección 0600. (En programas en lenguaje ensamblador, por cierto, los números hexadecimales suelen ir precedidas signos de dólar ($) para que puedan distinguirse de los números decimales.

Por lo tanto, en un programa de lenguaje ensamblador, las direcciones hexadecimales 0600 y 0601 se escriben $0600 y $0601.

Ahora vamos a suponer, sólo para fines ilustrativos, que usted necesita almacenar un número de 16 bits en dos direcciones de 8 bytes en la memoria RAM del computador ($0600 y $0601 por ejemplo), mientras ejecuta un programa en Basic. Dado que los programas en BASIC se escriben con números decimales ordinarios, lo primero que habría que hacer es convertir las direcciones con las que se va a trabajar, $0600 y $0601, en números decimales. Usted puede hacer esto de distintas maneras. Puede buscar los equivalentes decimales de $0600 y $0601 en una tabla de conversión decimal / hexadecimal. O también puede calcular las conversiones necesarias a mano. O también podría realizarlas con la ayuda de un programa. No importa cómo se las arregló para hacer las conversiones. Sin embargo, lo que acabaría descubriendo es que el número hexadecimal $0601 equivale al decimal 1537. Una vez que lo deduzca, podrá almacenar el valor de 16-bits en las direcciones $0600 y $0601 utilizando la siguiente rutina en Basic:

Rutina para almacenar números de 16 bits en la RAM

```
10 PRINT "INGRESE UN ENTERO POSITIVO"
20 PRINT "DE 0 A 65.535"
30 INPUT X
40 LET HIBYTE=INT(X/256)
50 LET LOBYTE=X-HIBYTE*256
60 POKE 1536,LOBYTE
70 POKE 1537,HIBYTE
80 END
```

Captura de pantalla

Ahora supongamos que usted quiere recuperar un número de 16 bits almacenado en la memoria RAM de su computador; por ejemplo, el número almacenado en la memoria de las direcciones $0600 y $0601 en el programa anterior. Y supongamos una vez más, que

quería hacer eso, mientras se ejecuta un programa en BASIC. Su rutina podría ser algo como esto:

Recuperación de números de 16 bits desde la RAM

```
10 LET X=PEEK(1537)*256+PEEK(1536)
20 PRINT X
```

Captura de pantalla

Conversión de números binarios a números decimales

Dado que los programadores de lenguaje ensamblador trabajan con tres diferentes tipos de números (decimal, hexadecimal y binario), a menudo se encuentran con que es necesario realizar conversiones entre una base numérica y otra. No es muy difícil de convertir un número binario a un número decimal. En un número binario, el bit más a la derecha representa 2 a la potencia de 0. El siguiente bit a la izquierda representa 2 a la potencia de 1, el siguiente representa 2 a la potencia de 2, y así sucesivamente. Los dígitos en un número binario de 8 bits se numeran del 0 al 7, partiendo desde el dígito de más a la derecha. A menudo se refiere al bit de más a la derecha como el menos importante (o LSB - Least Significant Bit). Representa el 2 a la potencia de 0, o sea, el número 1. Y el bit más a la izquierda - a menudo llamado el bit más significativo (o MSB - Most Significant Bit) - es igual a 2 a la potencia de 7, o sea, 128. He aquí una lista de ecuaciones simples que ilustran lo que cada bit en un número binario de 8 bits significa:

```
Bit 0 = 2^0 = 1
Bit 1 = 2^1 = 2
Bit 2 = 2^2 = 4
Bit 3 = 2^3 = 8
Bit 4 = 2^4 = 16
Bit 5 = 2^5 = 32
Bit 6 = 2^6 = 64
```

```
Bit 7 = 2^7 = 128
```

Utilizando la tabla de arriba, es fácil convertir cualquier número binario de 8 bits a su equivalente decimal. En lugar de escribir el número de abajo, de izquierda a derecha, escríbalo en una columna vertical, con el bit 0 en la parte superior de la columna y el bit 7 en la parte inferior. Luego se multiplican cada bit del número binario por el número decimal que representa. Luego sume el resultado de todas estas multiplicaciones. El total que se obtiene es el valor decimal del número binario. Supongamos, por ejemplo, que quiere convertir el número binario 00101001 en un número decimal. Aquí le mostramos cómo hacerlo:

```
1 X   1  =  1
0 X   2  =  0
0 X   4  =  0
1 X   8  =  8
0 X  16  =  0
1 X  32  =  32
0 X  64  =  0
0 X 128  =  0
________________
TOTAL    =  41
```

Según los resultados de este cálculo, el número binario 00101001 equivale al número decimal 41. Busque 00101001 o 41 en una tabla de conversión de binario a decimal o de decimal a binario, y verá que el cálculo es exacto. Y esta técnica de conversión funcionará con cualquier otro número binario. Ahora vamos a ir en la dirección contraria, y convertiremos un número decimal a binario. Y aquí está cómo lo haremos: Vamos a dividir el número por 2, y anotamos el cociente y el resto. Ya que se divide por 2, el cociente será un 1 o un 0. Así que vamos a escribir un 1 o un 0, según sea el caso. Luego tomaremos el cociente que tenemos, lo dividimos por dos, y escribimos el cociente abajo. Si hay un resto (un 1 o un 0), lo escribimos, también, justo debajo del primer resto. Cuando no queden más números que dividir, vamos a escribir todos los restos que tenemos, leyendo desde abajo hacia arriba. Lo que obtendremos entonces, por supuesto, es un número binario, un número compuesto de unos y ceros. Y ese número será el equivalente binario del número decimal con el que empezamos. Ahora vamos a probar esta técnica de conversión con el número decimal 117:

```
117/2 = 58 con resto 1
 58/2 = 29 con resto 0
 29/2 = 14 con resto 1
 14/2 =  7 con resto 0
  7/2 =  3 con resto 1
  3/2 =  1 con resto 1
  1/2 =  0 con resto 1
```

Según los resultados de este cálculo, el equivalente binario del número decimal 117 es 01110101. Y este resultado, como lo puede comprobar con una tabla de conversión de decimal a binario, también es correcto.

Conversiones De binario a Hexadecimal y de Hexadecimal a Binario

Es muy fácil convertir números binarios a sus equivalentes hexadecimales. Sólo tiene que utilizar esta tabla:

HEXADECIMAL	BINARIO
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

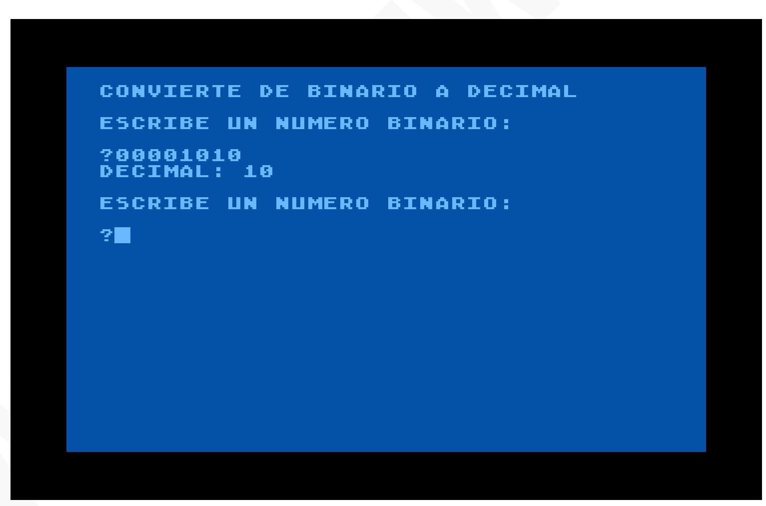
Para convertir un número hexadecimal de varios dígitos a binario, tome los dígitos hexadecimales y conviértalos cada uno por separado. Por ejemplo, el equivalente binario del número hexadecimal C0 es 1100 0000. El equivalente binario del número hexadecimal 8f2 es 1000 1111 0010. El binario equivalente al número hexadecimal 7A1B es 0111 1010 0001 1011. Y así sucesivamente. Para convertir los números binarios a números hexadecimales, utilice la tabla a la inversa. El número binario 1101 0110 1110 0101, por ejemplo, es equivalente al número hexadecimal D6E5.

Haciéndolo de la manera fácil

Aunque no es difícil convertir números binarios a hexadecimal y viceversa, lleva mucho tiempo hacerlo a mano, y cuando se programa en lenguaje ensamblador, tiene que hacer un montón de conversiones de binario a decimal y decimal a binario. Así que hay un montón de programas en BASIC para convertir números entre los sistemas de notación binaria y decimal. Y ahora le voy a regalar dos de ellos, absolutamente gratis. Aquí tiene uno para la conversión de números binarios a números decimales:

Conversión de Números Binarios a Números Decimales

```
10 DIM BN$(9),BIT(8),T$(1)
20 GRAPHICS 0
25 ? :? "CONVIERTE DE BINARIO A DECIMAL"
30 ? :? "ESCRIBE UN NUMERO BINARIO:":? :INPUT BN$
35 IF LEN(BN$)<>8 THEN 30
40 FOR L=1 TO 8
50 T$=BN$(L)
55 IF T$<>"0" AND T$<>"1" THEN 30
60 BIT(L)=VAL(T$)
70 NEXT L
75 ANS=0
80 M=256
90 FOR X=1 TO 8
100 M=M/2:ANS=ANS+BIT(X)*M
110 NEXT X
140 ? "DECIMAL: ";ANS
150 GOTO 30
```



Captura de pantalla

Y aquí hay un programa para la conversión de números decimales a números binarios:

Conversión de Números Decimales a Números Binarios

```
10 DIM BIN$(8),TEMP$(8),R$(1)
20 GRAPHICS 0
30 ? :? "CONVIERTE DE DECIMAL A BINARIO"
40 ? :? "INGRESE EL NUMERO (0 AL 255):":? :TRAP 40:INPUT NR
50 IF NR-INT(NR)<>0 THEN 40
60 IF NR>255 OR NR<0 THEN 40
```

```
70 FOR L=1 TO 8
80 Q=NR/2
90 R=Q-INT(Q)
100 IF R=0 THEN R$="0":GOTO 120
110 R$="1"
120 TEMP$(1)=R$:TEMP$(2)=BIN$:BIN$=TEMP$
130 NR=INT(Q)
140 NEXT L
150 ? "BINARIO: ";BIN$
160 TRAP 40000
170 GOTO 40
```

Captura de pantalla

Conversión de Decimal a Hexadecimal

La conversión de decimal a hexadecimal es un proceso complejo que es mejor realizarlo usando un computador o una calculadora especial. Texas Instruments ofrece una calculadora llamada The Programmer que puede realizar conversiones de decimal a hexadecimal en un segundo, y también puede sumar, restar, multiplicar y dividir ambos números decimales y hexadecimales. Muchos diseñadores de programadores en lenguaje ensamblador usan The Programmer de Texas Instruments (TI), o cualquier otra calculadora similar, y tienen dificultades para manejarse sin ella. En caso de que no pueda conseguir una calculadora de programadores ahora mismo, aquí hay un programa en BASIC de Atari que convierte números decimales a números hexadecimales y viceversa inversa. Hay otro programa está disponible en la página H-18 del manual de referencia BASIC que viene con su cartucho BASIC de Atari.

Programa para convertir de Decimal a Hexadecimal y de Hexadecimal a Decimal

```
10 REM
20 REM CONVIERTE DE HEXADECIMAL A DECIMAL O VICEVERSA
```

```
30 REM
40 REM EL CAMINO RAPIDO
50 REM NO UTILICE MATEMATICAS
60 REM
70 DIM H$(40),A$(40)
80 REM
90 PRINT
100 PRINT
110 PRINT
120 PRINT "CONVIERTE DE HEXADECIMAL A DECIMAL O  VICEVERSA"
130 PRINT
140 PRINT "H) HEX A DEC"
150 PRINT "D) DEC A HEX"
160 PRINT
180 INPUT A$
190 IF A$="H" THEN 220
200 IF A$="D" THEN 400
210 GOTO 100
220 REM
230 REM CONVIERTE DE HEXADECIMAL A DECIMAL
240 REM
250 PRINT "INGRESE NUMERO HEX :";
260 INPUT H$
270 REM
280 D=0
290 S=1:REM MULTIPLICADOR POSICIONAL
295 REM PASAR POR LA CADENA
300 FOR L=LEN(H$) TO 1 STEP -1
310 A$=H$(L,L)
320 REM
330 REM CONVIERTE "0"-"F" A 0-15
340 N=ASC(A$)-48:IF N>9 THEN N=N-7
345 IF N<0 OR N>15 THEN 90
350 D=D+N*S
360 S=S*16
370 NEXT L
380 PRINT "HEX ";H$;"= DEC ";D
390 GOTO 90
400 REM
410 REM CONVIERTE DE DECIMAL A HEXADECIMAL
420 REM
430 PRINT "INGRESE NUMERO DEC :";
440 INPUT D
450 REM
460 REM EN PRIMER LUGAR ENCONTRAMOS EL DIGITO MAS ALTO
```

```
470 REM
480 S=16
490 X=2
500 IF S<D THEN X=X+1:S=S*16:GOTO 500
505 PRINT "DECIMAL ";D;" = HEX ";
510 T=D
520 FOR L=X TO 1 STEP -1
530 N=INT(T/S)
540 PRINT CHR$(48+N+7*(N>9));:REM CONVIERTE DEC A HEX 0-F
550 T=T-N*S:S=S/16
560 NEXT L
570 PRINT
580 GOTO 90
```

Captura de pantalla

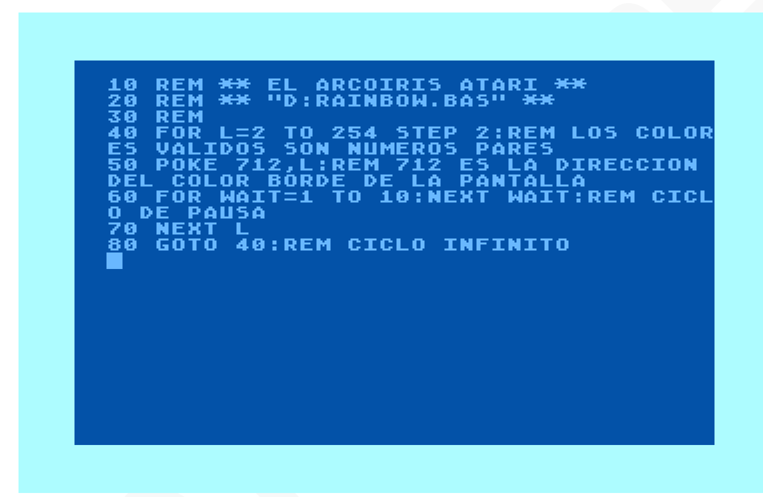
Así concluye nuestro curso intensivo en bits, bytes y números binarios. Este fue un capítulo importante porque es un requisito previo para el capítulo 3, que a su vez es un requisito previo para el capítulo 4, donde tendré la oportunidad de realmente comenzar a escribir algunos programas en lenguaje ensamblador.

Algo para progresar

Mientras tanto, aquí hay otro programa BASIC que puede ayudarle a sentir que se está comunicando con su Atari. Utiliza un bucle infinito para recorrer todos los colores y matices que su computador puede generar, cargando cada uno de ellos en el registro de memoria que controla la zona del borde de la pantalla de video. En el capítulo 9, Programando Bit a Bit, aprenderá cómo hacer esto mismo truco utilizando el lenguaje ensamblador.

PROGRAMA DE EJEMPLO N°2
El Arcoíris ATARI

```
10 REM ** EL ARCOIRIS ATARI **
20 REM ** "D:RAINBOW.BAS" **
30 REM
40 FOR L=2 TO 254 STEP 2:REM LOS COLORES VALIDOS SON NUMEROS
PARES
50 POKE 712,L:REM 712 ES LA DIRECCION DEL COLOR BORDE DE LA
PANTALLA
60 FOR WAIT=1 TO 10:NEXT WAIT:REM CICLO DE PAUSA
70 NEXT L
80 GOTO 40:REM CICLO INFINITO
```

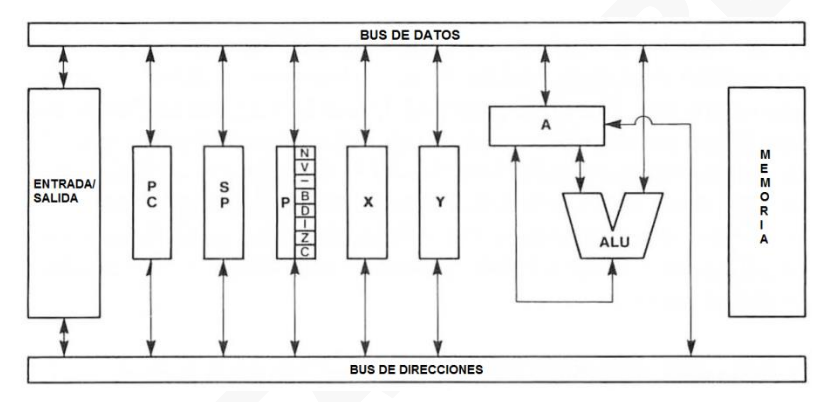


Captura de pantalla

¡Digite este programa en su computador, ejecútelo, y disfrute el espectáculo! Luego seguiremos con el Capítulo 3.

Capítulo Tres
Dentro del 6502

En este capítulo vamos a mirar dentro de su computador Atari para ver cómo funciona. Entonces será capaz de encontrar el camino de entrada a su computador y finalmente empezar a programar en lenguaje ensamblador. Como explicamos en el capítulo 1, cada computador tiene tres partes principales: Una Unidad Central de Proceso (CPU - Central Processing Unit), memoria (dividida en RAM y ROM), y dispositivos de entrada y salida (tales como teclado, monitor de video, grabadoras de casette, y unidades de disco). En un micro-computador, todas las funciones de una CPU están contenidas en la Unidad de Microprocesador (a veces abreviado MPU - Micro Processor Unit). La MPU de su Atari es un microprocesador 6502.



El microprocesador 6502 de Atari

El microprocesador 6502 contiene siete partes principales: una Unidad Aritmética Lógica (ALU - Arithmetic Logical Unit) y seis registros direccionables. Los datos se mueven dentro del chip 6502, y entre el 6502 y los otros componentes de su computador a través de líneas de transmisión llamadas buses. Hay dos tipos de buses en un computador Atari: un bus de datos de 8-bits y un bus de direcciones de 16-bits. El bus de datos es utilizado para pasar datos de 8 bits y bytes de instrucción desde un registro del 6502 a otro, y también para pasar datos e instrucciones entre el 6502 y la memoria de su computador. El bus de direcciones se utiliza para estar al tanto de las direcciones de memoria de 16 bits de su computador: las direcciones de las que vienen instrucciones y datos, y las direcciones a las que van instrucciones y datos.

La Unidad Aritmética Lógica

El componente más importante de su equipo es el chip 6502. Y la parte más importante de chip 6502 es su Unidad Aritmética Lógica (ALU - Arithmetic Logical Unit). Cada vez que su equipo realiza un cálculo o una operación lógica, la ALU es la que realiza todo el trabajo. La ALU puede realizar sólo dos tipos de cálculos: sumas y restas. Problemas de división y multiplicación también pueden ser resueltos por la ALU, pero sólo en forma de

secuencias de sumas y restas. La ALU también puede comparar valores, restando un valor de otro, y luego observando los resultados de la resta. La ALU del chip 6502 tiene dos entradas y una salida. Cuando dos números se van a sumar, restar o comparar, uno de los números es alimentado a la ALU a través de una de sus entradas, y el otro número es alimentado a través de la otra entrada. La ALU se encarga de efectuar el cálculo solicitado, y pone el resultado en un bus de datos para que pueda ser transportado a donde sea necesario en el programa.

En los diagramas del chip 6502, la ALU es a menudo representada por un contenedor en forma de V. Los brazos de la V son las entradas de la ALU, y la parte inferior de la V es la salida de la ALU. Cuando se necesita realizar un cálculo o una operación lógica en la ALU, uno de los datos y el operando (adición o sustracción) son depositados en una de las entradas de la ALU (un brazo de la V). El otro dato se deposita en la otra entrada (el otro brazo la V). Cuando el cálculo se realice, el resultado es expulsado a través de la salida de la ALU (la parte inferior de la V).

El acumulador

La ALU no trabaja sola, sino que lleva a cabo todas sus operaciones con la ayuda de un registro del 6502 llamado el Acumulador (abreviado "A" - Accumulator). Cuando la ALU es llamada para sumar o restar dos números, uno de los números es puesto en un bus de datos y luego enviado a una de las entradas de la ALU, junto con un operando. El otro número se encuentra en el acumulador. Cuando el bus que transporta al número y su operando vuelca su contenido en la ALU, se pone el número que contiene el acumulador en el bus de datos y se envía ese número a la ALU. Cuando la ALU ha llevado a cabo el cálculo solicitado, deposita el resultado del cálculo en el acumulador.

Un ejemplo

Supongamos, por ejemplo, que necesita que su computador sume 2 más 2, y luego coloque el resultado del cálculo en una posición de memoria determinada. Se podría utilizar una rutina de lenguaje ensamblador como ésta:

```
LDA #02
ADC #02
STA $CB
```

La primera instrucción en esta rutina, "LDA", significa "cargue el acumulador - LoaD the Accumulator" (con el valor que viene a continuación). En este caso, ese valor es 2. El símbolo "#" delante del 2 significa que el 2 debe ser interpretado como un número literal, no como la dirección de una posición de memoria en su computador.

La segunda instrucción de la rutina, ADC, significa "Suma con acarreo - ADd with Carry". En esta suma no hay ningún número que deba ser acarreado, por lo tanto la parte

"acarreo" de la instrucción no tiene ningún efecto en este caso, y todo lo que hace la instrucción ADC es sumar 2 más 2.

La tercera y última instrucción en nuestra rutina, STA, significa "almacenar el contenido del acumulador - STore the contents of the Accumulator" (en la dirección de memoria que viene a continuación).

Como puede ver, la dirección de memoria a continuación de la instrucción STA es $CB, el hexadecimal correspondiente al número decimal 203. Dado que no hay signo "#" delante del número hexadecimal $CB, el ensamblador no interpretará $CB como un número literal. En cambio, $CB será interpretado como una dirección de memoria, que es como debe ser interpretado un número en lenguaje ensamblador cuando éste no es un número literal. (Por cierto, si quiere que su ensamblador interprete $CB como un número literal, usted tendría que escribir "#$CB". Cuando un símbolo "#" y un signo de dólar "$" aparecen antes de un número éste se interpreta como un número hexadecimal literal.) Sin embargo, si la tercera línea de nuestra rutina fuera STA #$CB, tendríamos un error de sintaxis. Esto porque STA (almacenar el contenido del acumulador en...) es una instrucción que tiene que ser seguida por un valor que pueda ser interpretado como una dirección de memoria, y no por un número literal.

Otros Cinco Registros

Además del acumulador, el procesador 6502 tiene otros cinco registros. Estos son el registro X, el registro Y, el Contador del Programa ("Program Counter"), el Puntero de la Pila ("Stack Pointer"), y el Registro de Estado del Procesador ("Processor Status Register"). He aquí un breve resumen de las funciones de cada uno de estos registros.

Los Otros Registros del 6502

• El Registro X (abreviado "X") es un registro de 8 bits que se utiliza a menudo para el almacenamiento temporal de datos durante la ejecución de un programa. Sin embargo, el registro X tiene una característica especial: también puede ser incrementado y disminuido en su valor con el par de instrucciones de un byte INX y DEX en lenguaje ensamblador, y es por lo tanto usado a menudo como un registro índice, o contador, en los bucles y durante la ejecución de instrucciones de programas de lectura de información.

• El Registro Y (abreviado "Y") es un registro de 8 bits, y también puede ser incrementado y disminuido en su valor con el par de instrucciones de un byte INY y DEY. Así que el registro Y, tal como el registro X, se utiliza para el almacenamiento de datos y como contador.

• El Contador del Programa (abreviado "PC - Program Counter") es un par de registros de 8 bits que se utilizan juntos como si fueran un solo registro de 16 bits. A veces a los dos registros de 8 bits que se combinan para formar el Contador de Programa se les

denomina "Contador del Programa-Bajo (PCL - Program Counter-Low)" y "Contador del Programa-Alto (PCH - Program Counter-High)". El Contador del Programa siempre contiene la dirección de memoria de 16 bits de la siguiente instrucción a ser ejecutada por el procesador 6502. Cuando la instrucción se ha ejecutado, la dirección de la siguiente instrucción se cargará en el contador de Programa.

• El Puntero de la Pila (abreviado "S" o "SP") ("Stack Pointer") es un registro de 8 bits que siempre contiene la dirección del elemento superior de un bloque de memoria RAM llamado la Pila de Hardware ("Hardware Stack"). Por lo general, a la Pila de Hardware se le conoce simplemente como "La Pila". Es un segmento especial de memoria en la que los datos se almacenan temporalmente durante la ejecución de un programa. Más adelante vamos a entrar en más detalles acerca de cómo funciona la Pila.

• El Registro de Estado del Procesador ("Processor Status Register", por lo general llamado simplemente el "Registro de Estado," pero abreviado "P"), es un registro de 8 bits que está al tanto de los resultados de las operaciones que han sido realizadas por el procesador 6502.

El Registro de Estado del Procesador

El Registro de Estado del Procesador (P) es un poco diferente de los otros registros en el microprocesador 6502. No se utiliza para almacenar números de 8 bits ordinarios, como los otros registros del 6502. En su lugar, los bits de este registro son banderas ("flags") que están al tanto de información importante. Cuatro de los bits del Registro de Estado se llaman Banderas de Estado. Estos son la Bandera de Acarreo ("Carry Flag - C"), la Bandera de Desbordamiento ("Overflow Flag - V"), la Bandera de Negativo ("Negative Flag - N"), y la Bandera de Cero ("Zero Flag - Z"). Estos cuatro indicadores se utilizan para hacer un seguimiento de los resultados de las operaciones que se llevan a cabo en los otros registros del procesador 6502.

Tres de los otros bits del registro P, llamados Banderas de Condición, se utilizan para determinar ciertas condiciones existentes en un programa. Estos tres bits son la Bandera de Deshabilitación de Interrupción ("Interrupt Disable Flag - I"), la Bandera de Ruptura ("Break Flag - B"), y la Bandera de Modo decimal ("Decimal Mode Flag - D"). El octavo bit del registro de estado no se utiliza.

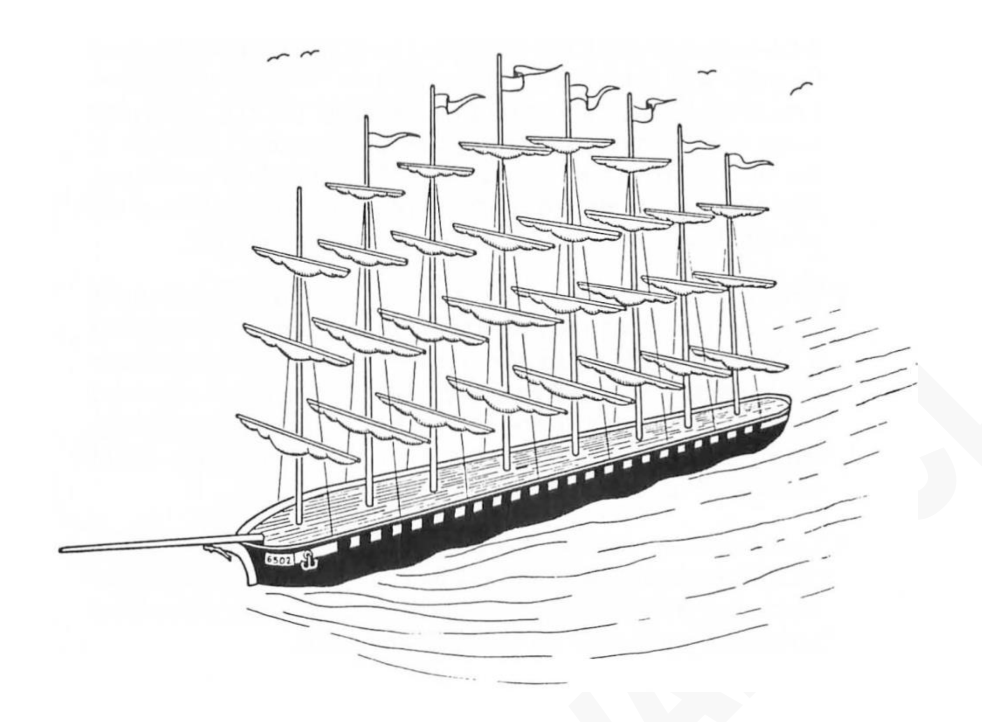


Diagrama del Registro de Estado del Procesador

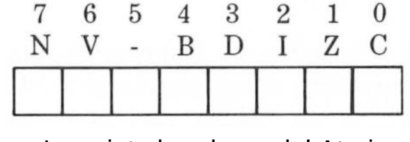
El Registro de Estado puede ser imaginado como una caja rectangular que contiene seis compartimentos cuadrados. Cada "compartimiento" de la caja es en realidad un bit, y cada bit es utilizado como una bandera.

Si un bit es un "1" en vez de un "0", entonces se dice que es una bandera que ha sido activada ("set").

Si un bit es un "0" en vez de un "1", entonces se dice que es una bandera que ha sido desactivada ("cleared").

Los bits en el Registro de Estado de 6502, al igual que los bits en todos los registros de 8 bits, habitualmente son numerados del 0 al 7. El bit más a la derecha es el bit 0. El bit más a la izquierda es el bit 7.

Una ilustración del Registro de Estado del procesador



Las siete banderas del Atari

He aquí una lista completa de las banderas en el Registro de Estado del Procesador 6502, y una explicación de lo que significa cada una de ellas.

Bit 0 (El bit de más a la derecha)
La Bandera de Acarreo (C)

Como vimos en el capítulo 2, no es fácil hacer aritmética de 16 bits en un chip de 8 bits como el 6502. Cuando el chip 6502 requiere realizar una operación de suma con un número superior a 255, o si el resultado de un cálculo es mayor que 255, se tiene que escribir un programa que divida cada número en segmentos de 8 bits para poder ser procesados, y posteriormente juntar los números nuevamente. Esta clase de matemática de cortar y pegar, como usted podrá imaginar, implica una gran cantidad de acarreos (si se están realizando sumas) y préstamos (cuando el 6502 realiza una resta). La bandera de Acarreo del registro P del 6502 está al tanto de todos estos acarreos y préstamos.

Si una suma implica un acarreo, la bandera de Acarreo se activa automáticamente. Y si una resta implica un préstamo, la bandera de acarreo estará al tanto de eso también. Dado que la bandera de acarreo se activa y desactiva casi constantemente como consecuencia de los acarreos y préstamos de sumas y restas, es siempre una buena idea desactivarlo antes de realizar una suma, y activarlo antes de realizar una resta. De lo contrario, existe la posibilidad de que su cálculo se corrompa debido a los resultados de las sumas y restas realizadas anteriormente.

Además de estar al tanto de las sumas y restas, la bandera de Acarreo del registro P también es utilizada en operaciones de comparación de valores, y en operaciones de desplazamiento ("shift") y rotación de bits para comprobar, comparar y manipular bits específicos de un número binario. Hablaremos de las comparaciones de números y las operaciones sobre bits en capítulos posteriores. Por ahora, es más importante recordar que la instrucción en lenguaje ensamblador para desactivar el bit de Acarreo del registro P es CLC, que significa Limpiar Acarreo ("CLear Carry"), y que la instrucción para activar el bit es SEC, que significa Activar Acarreo ("SEt Carry").

Bit 1 (El segundo bit de la derecha)
La Bandera Cero (Z)

Cuando el resultado de una operación aritmética o lógica es cero, la bandera Cero del Registro de Estado se activa automáticamente. Sumas, restas y operaciones lógicas pueden resultar en cambios en el estado de la bandera Cero. Si una posición de memoria o un registro de índice son decrementados hasta llegar a cero, se desactivará la bandera Cero. Una convención irónica del 6502 es que cuando el resultado de una operación es cero, la bandera Cero tendrá el valor 1 (activado), y cuando el resultado de una operación es distinto de cero, la bandera Cero tendrá el valor 0 (desactivado). Es importante entender este concepto, ya que sería fácil suponer que la bandera Cero opera de manera

contraria. No hay instrucciones en lenguaje ensamblador para activar o desactivar la bandera Cero. Es estrictamente un bit "de solo lectura", por lo que no hay instrucciones para modificarlo.

Bit 2 (El tercer bit de la derecha)
La Bandera de Deshabilitación de Interrupción (I)

Los programas Atari pueden contener interrupciones, es decir, instrucciones que detienen temporalmente las operaciones para que se puedan realizar otras. Unas se llaman Interrupciones Enmascarables ("Maskable Interrupt") porque se puede prevenir que se ejecuten por medio de la utilización de instrucciones de ocultación ("masking") dentro de un programa. Las otras se llaman No-Enmascarables ("nonmaskable") porque no se puede evitar que se ejecuten, sin importar lo que se pueda hacer al respecto. Cuando desee desactivar una interrupción enmascarable, puede hacerlo usando la bandera de Deshabilitación de Interrupción del registro P. Cuando se activa esta bandera, no se permitirá la ejecución de interrupciones enmascarables. Cuando está desactivada, se permite la ejecución. La instrucción en lenguaje ensamblador para desactivar la bandera de interrupciones es CLI ("CLear Interrupt"). La instrucción para activar la bandera de interrupciones es SEI ("SEt Interrupt").

Bit 3 (El cuarto bit de la derecha)
La Bandera de Modo Decimal (D)

El procesador 6502 normalmente funciona en modo binario, es decir, usando números binarios estándar del tipo que se discutieron en el capítulo 2. Pero el 6502 también puede operar en lo que se conoce como “modo decimal codificado en binario”, o modo BCD ("Binary-Coded Decimal"). Para poner al 6502 en modo BCD, se tiene que activar la bandera Modo Decimal del Registro de Estado del 6502. La aritmética BCD es más lenta que la aritmética binaria simple, y además consume más memoria. Sin embargo, sus resultados, a diferencia de las de la aritmética binaria simple, siempre son 100 por ciento exactos. Por lo tanto, es de uso frecuente en los programas y rutinas en la que la precisión es más importante que la velocidad o la eficiencia de la memoria.

Un ejemplo de un programa que utiliza aritmética BCD es el intérprete Atari BASIC. En el BASIC de Atari los números se almacenan como números BCD de 6 bytes, y toda la aritmética se realiza con la aritmética BCD. Debido a esta característica, Atari BASIC es un poco lento. Sin embargo, sus cálculos dan resultados precisos. Otra ventaja de los números BCD es que son más fáciles de convertir a números decimales en comparación a los números binarios. Así los números BCD a veces son usados en programas que requieren la visualización instantánea de números en un monitor de video.

Vamos a discutir el modo BCD en el capítulo 10, Matemáticas del Lenguaje Ensamblador. Por el momento, es suficiente decir que cuando se activa la bandera Modo Decimal del

Registro de Estado, el chip 6502 llevará a cabo todas sus operaciones matemáticas usando números BCD. La Aritmética BCD es algo que usted rara vez deseará usar en un computador Atari, por lo que normalmente querrá asegurarse de que la bandera Decimal esté desactivada cuando el equipo esté realizando operaciones matemáticas. La instrucción en lenguaje ensamblador que desactiva la bandera decimal es CLD ("CLear Decimal"), y la instrucción que activa la bandera es SED ("SEt Decimal").

Bit 4 (El quinto bit de la derecha)
La Bandera de Detención ("Break") (B)

La bandera de detención se activa mediante la instrucción especial BRK ("BReaK"). Diseñadores de programas a menudo usan la instrucción de detención en la fase de depuración ("debug") de programas en lenguaje ensamblador. Cuando se utiliza esta instrucción y la bandera de detención ha sido activada, se llevan a cabo unas operaciones de marcado de errores ("Error Flagging") y el control del equipo vuelve al programador. La instrucción de detención es una herramienta de depuración muy compleja, y no vamos a entrar en muchos detalles al respecto en este libro. Pero puede aprender más sobre esta instrucción en algunos de los textos avanzados de programación del 6502 que se nombran en la bibliografía.

Bit 5 (El sexto bit de la derecha)
[Bit sin uso]

Por alguna razón, los microprogramadores que diseñaron el Registro de Estado del 6502 dejaron un bit sin usar. Este es el único.

Bit 6 (El segundo bit de la izquierda)
La Bandera de Desbordamiento (V)

La Bandera de Desbordamiento se utiliza para detectar un desbordamiento del bit 6 de un número binario. Si no sabe lo que esto significa, no se preocupe. La bandera de desbordamiento se usa principalmente en aritmética avanzada del 6502, en particular cuando se está realizando aritmética binaria para estar al tanto de los cambios en los signos más y menos de números con signo ("signed numbers"). Como programador de lenguaje ensamblador de Atari, raramente, si es que alguna vez, tendrá la oportunidad de utilizar la Bandera de Desbordamiento. Sin embargo, vamos a discutirlo en mayor detenimiento en el capítulo 10, Matemáticas del Lenguaje Ensamblador. Por ahora, vamos a agregar que la instrucción en lenguaje ensamblador para desactivar la Bandera de Desbordamiento es CLV ("CLear oVerflow"). No hay ninguna instrucción para activar esta bandera.

Bit 7 (El bit de la izquierda)
La Bandera Negativo

La Bandera Negativo se activa cuando el resultado de una operación es negativo, y se desactiva cuando el resultado de una operación es cero. Se utiliza a menudo en operaciones de números con signo, y también tiene otros usos que se discutirán en capítulos posteriores. No hay instrucciones para activar o desactivar la Bandera Negativo. No hay necesidad para ello, ya que la bandera se utiliza solamente para propósitos de comprobación.

Su Gran Oportunidad

Ahora que sabe lo que sucede dentro del procesador 6502 de su computador, ya está preparado para escribir y ejecutar su primer programa en lenguaje ensamblador. Tendrá la oportunidad de hacerlo precisamente en el próximo capítulo. Pero antes, aquí está la oportunidad de ejecutar otro programa extra que contiene algunos valores en lenguaje de máquina que pueden ser ingresados por medio de POKEs dentro de un programa en BASIC. Esta rutina, Programa Extra N º 3, convertirá el teclado de su computador en un teclado musical. Digítelo y ejecútelo, y luego veremos cómo funciona.

PROGRAMA EXTRA N° 3
“D:SOUNDOFF.BAS”

```
10 REM "D:SOUNDOFF.BAS"
20 INKEY=35:FREQ=53760
30 SOUND 0,0,0,0:GRAPHICS 0:OPEN #1,4,0,"K:"
35 GET #1,K:IF K=32 THEN SOUND 0,0,0,0:PRINT CHR$(K):GOTO
INKEY:REM 32 ES UN ESPACIO
40 IF K<64 OR K>122 THEN GOTO INKEY:REM NO ES LETRA
50 IF K>90 AND K<97 THEN GOTO INKEY:REM NO ES LETRA
60 IF K>90 THEN K=K-32:REM CAMBIO DE MINUSCULAS A MAYUSCULAS
70 TONE=K-64:IF TONE<1 OR TONE>7 THEN GOTO INKEY:REM NO A,B,
C,D,E,F o G
80 PRINT CHR$(K);:GOSUB 1000
90 ON TONE GOTO 100,200,300,400,500,600,700
100 POKE FREQ,145:GOTO INKEY:REM A
200 POKE FREQ,134:GOTO INKEY:REM B
300 POKE FREQ,121:GOTO INKEY:REM C
400 POKE FREQ,108:GOTO INKEY:REM D
500 POKE FREQ,96:GOTO INKEY:REM E
600 POKE FREQ,91:GOTO INKEY:REM F
700 POKE FREQ,81:GOTO INKEY:REM G
1000 SOUND 0,255,10,8:RETURN :REM CONFIGURA REGISTRO DE AUDIO
PARA LAS NOTAS A TOCAR
```

Cómo funciona

En las líneas 30 y 35 de este programa los registros de audio de su computador se inicializa en cero y se configura un bucle o ciclo para imprimir los caracteres en la pantalla. En las líneas 40 y 70 se llevan a cabo algunas verificaciones para ver si los caracteres que se han digitado son notas válidas: A, B, C, D, E, F o G. Para hacer que el programa funcione sin problemas, en la línea 60 automáticamente se convierte a mayúsculas cualquier carácter que haya sido escrito en minúsculas. En la subrutina en la línea 1000 los registros de audio de su computador se preestablecen para reproducir las notas de la A a la G, cuando las teclas correspondientes son presionadas, a menos que dicha tecla sea un espacio. Si la tecla es un espacio, todos los registros de audio se apagan, se imprime un espacio en la pantalla, y se le ordena al computador que espere la siguiente tecla presionada (lo que ocurre en la línea 35).

Las instrucciones de lenguaje de máquina en el programa están en líneas de la 100 a la 700. En estas líneas, un registro de memoria en su computador (llamado FREQ en este programa) es asignado con un valor que equivale a una nota musical. Cada vez que cambia el valor, la nota que se está reproduciendo cambia. Este es un programa muy simple que ni siquiera comienza a explorar las elaboradas capacidades de sonido de los computadores Atari.

Las formas en que este programa pueda ampliarse sólo están limitadas por la imaginación del usuario. Puede hacer el programa un poco más complicado, agregándole la capacidad de reproducción de sostenidos y bemoles (con la tecla de control, por ejemplo), y también se puede añadir más octavas (por ejemplo, con la tecla de mayúsculas, las teclas de números, o combinaciones de las teclas Shift y Control). ¡Podría cambiar los colores de la pantalla a medida que las notas cambian o podría almacenar las melodías en la memoria de su computador y guardarlas en un disco, y podría incluso ser capaz de encontrar una manera de tocar acordes! Puede hacer todas estas cosas en BASIC, si lo desea o, si fuera más ambicioso, podría ir más allá del BASIC y seguir aprendiendo más acerca de lenguaje ensamblador de su Atari. Lo que nos lleva al Capítulo 4. Siga leyendo.

Capítulo Cuatro
Escribiendo un programa en lenguaje ensamblador

Aquí está, por fin, el programa de lenguaje ensamblador que le prometimos: un programa que puede digitar, ensamblar y ejecutar sin tener que depender de ningún comando del Lenguaje BASIC. Este programa fue escrito en un computador Atari 800 viejo y lleno de cicatrices de batalla, usando un antiguo cartucho Atari Assembler Editor. Pero el programa se ejecutará en cualquier computadora Atari y, como todos los programas de este libro, es totalmente compatible con los cartuchos Atari Assembler Editor y el nuevo y más rápido ensamblador MAC/65 fabricado por OSS (Optimized Systems Software Inc., de San José, California).

Con unas pequeñas pero criticas modificaciones, tales como eliminar los números de línea y cambiar en la línea 40 la directiva "*= $0600" por "org $0600", este programa, y todos los demás en este libro pueden ser convertidos para que funcionen con el cartucho Atari Macro Assembler y su editor de texto. Pero a menos que usted ya sea un programador de lenguaje ensamblador, recomiendo encarecidamente que digite, ensamble, depure y edite los programas de este libro usando un ensamblador similar al MAC/65 o al cartucho Atari Assembler Editor, que son los que fueron utilizados para escribir dichos programas.

Aquí está el programa con el que vamos a trabajar en este capítulo. Como puede ver, se trata de un programa muy simple para sumar 2 más 2. Echemos una mirada en detalle, y veamos cómo hace lo que se supone que tiene que hacer.

UN PROGRAMA DE SUMAS DE 8 BITS
(ADDNRS.SRC)

```
10   ;
20   ;ADDNRS.SRC
30   ;
40               *=$0600
50   ;
60               CLD
70   ADDNRS      CLC
80               LDA #2
90               ADC #2
100              STA $CB
110              RTS
120              .END
```

```
ASM
                 10 ;
                 20 ;  ADDNRS.SRC
                 30 ;
0000             40             *=    $0600

                 50 ;
0600 D8          60 ADDNRS      CLD
0601 18          70             CLC
0602 A902        80             LDA   #2
0604 6902        90             ADC   #2
0606 85CB        0100           STA   $CB
0608 60          0110           RTS
0609             0120           .END

EDIT
```

Captura de pantalla de ADDNRS.SRC

Mire de cerca este programa y verá que los números que serán sumados, 2 y 2, están en la línea 80 y 90. Luego de que el programa realiza la suma, almacena el resultado del cálculo en la dirección de memoria $CB (o 203 en decimal). Eso pasa en la línea 100. Algunas instrucciones del programa pueden parecer conocidas; hemos visto la mayoría de ellas en los capítulos anteriores. Sin embargo, hay algunos elementos en el programa que se ven aquí por primera vez. Estos incluyen los puntos y comas en las primeras líneas del programa, la directiva "*=" en la línea 40, y la directiva ".END" en la línea 120.

Espaciamiento

Como se puede ver más claramente en el "diagrama ampliado" que sigue, el listado del código fuente del programa se divide en cuatro campos o columnas. Si cada campo tuviera un encabezado, y si el listado del programa no estuviera todo apretujado, esto es lo que se vería:

Un programa de sumas de 8 bits (ADDNRS.SRC)
(Listado columna a columna)

NOMBRE DE CAMPO	NUMERO DE LINEA	ETIQUETA	OP CODE	OPERANDO	COMENTARIO
	10	;			
	20	;ADDNRS.SRC			
	30	;			
	40		*=	$0600	
	50	;			
	60	ADDNRS	CLD		
	70		CLC		
	80		LDA	#2	
	90		ADC	#2	
	0100		STA	$CB	
	0110		RTS		
	0120		.END		

Sólo un ejemplo

El listado anterior es tan sólo una ilustración de cómo se vería el listado de un código fuente si sus cuatro campos (números de línea, etiquetas, op codes y comentarios) estuvieran claramente separados por columnas. En realidad, nadie escribe listados de código fuente de esta manera.

Cuando se escribe un listado de código fuente usando el MAC/65 o el cartucho Atari Assembler Editor, la forma habitual de hacerlo es digitando sus campos en un formato que es más bien hacinado: tan apretujado que las columnas no están alineadas para nada. Sin embargo, una vez que llegue a tener un poco de práctica en la escritura de código de esta manera, verá que se convierte en algo natural.

Estas son las reglas:

Los números de línea

Cuando se escribe un listado de código fuente mediante MAC/65 o el cartucho Atari Assembler Editor, a cada sentencia, o línea, se le debe asignar un número de línea. Los números de línea del programa se escriben a la izquierda, tal como en los programas en BASIC. Los números de línea no son realmente necesarios en los programas en lenguaje ensamblador, y algunos ensambladores no los requieren. El paquete Atari Macro Assembler and Text Editor, por ejemplo, no requiere el uso de los números de línea y no ensamblará un programa que los incluya. Pero MAC/65 y el cartucho Atari Assembler Editor usan números de línea, por lo que también los usaremos, al menos por ahora.

Los números de línea en la primera columna de nuestro programa de sumas aumentan en incrementos de 10, al igual que los números en un programa típico de BASIC. No tienen que ser escritos de esa manera, pero por lo general es así, como es el caso de BASIC. Los números de línea pueden ir de 0 a 65535.

Etiquetas

Las etiquetas, si se utilizan, ocupan el segundo campo en las sentencias del lenguaje ensamblador. Se debe dejar exactamente un espacio, no dos, entre un número de línea y una etiqueta. Si escribe una etiqueta dos o más espacios después de un número de línea, o si se utiliza el tabulador para llegar al campo etiqueta, su programa no funcionará. En programas de lenguaje ensamblador, las etiquetas se utilizan para identificar las primeras líneas de las rutinas y subrutinas. Nuestro programa para sumar 2 y 2, por ejemplo, tiene la etiqueta ADDNRS en la línea 60.

Dado que nuestro programa tiene una etiqueta, este podría ser utilizado como una subrutina en un programa más largo, y se podría acceder fácilmente por medio de su

etiqueta. Para invocarla se pueden usar varios tipos de instrucciones en lenguaje ensamblador. Por ejemplo, JSR ADDNRS (Salte a la subrutina que se encuentra en la etiqueta ADDNRS), BCS ADDNRS (Salte a la etiqueta ADDNRS cuando se desactive la bandera de acarreo), o JMP ADDNRS (saltar a la etiqueta ADDNRS). En el primer ejemplo, ADDNRS terminaría con la instrucción RTS (retorno de la subrutina) en la línea 100. RTS realiza el mismo tipo de trabajo en lenguaje ensamblador que la instrucción RETURN realiza en BASIC: señaliza el final de una subrutina, y el control vuelve de nuevo al cuerpo principal del programa. En los últimos ejemplos, se podría usar una segunda instrucción de salto en lugar de RTS para transferir el control.

Hablaremos de saltos, instrucciones de ramificación y subrutinas en mayor detalle en capítulos posteriores. Por ahora, lo más importante que debe recordar es que cuando se escribe un listado de código fuente en un ensamblador MAC/65 o un cartucho Atari Assembler Editor, se debe usar uno y sólo un espacio para separar cada etiqueta de su número de línea. El largo de una etiqueta puede ser desde un carácter hasta el largo que permita la sentencia (106 caracteres menos la cantidad de caracteres usados en el número de línea de la instrucción). La mayoría de los programadores usan etiquetas de entre tres y seis caracteres de longitud.

Mnemónicos de Código de Operación

Un Mnemónico de Código de Operación (u "op code") es sólo un nombre bonito para referirse a las instrucciones en lenguaje ensamblador. Hay 56 "op codes" en el conjunto de instrucciones del 6502, y son los únicos que pueden ser utilizados en las instrucciones del lenguaje ensamblador de Atari. "Op codes" tales como CLC, CLD, LDA, ADC, STA y RTS se escriben en el campo "op code" de los listados de código fuente de lenguaje ensamblador. Cuando se escribe un programa usando MAC/65 o el Atari Assembler Editor, cada "op code" que utilice debe comenzar con al menos dos espacios después del número de línea, o un espacio después de una etiqueta. Un "op code" colocado en el campo equivocado no se marcará como un error al escribir su programa, pero se marcará como un error cuando sea ensamblado.

El campo "op code" en un listado de código fuente se usa también para directivas y "pseudo ops", palabras y símbolos que se introducen en un programa como mnemotécnicos, pero que no son en realidad miembros del conjunto de instrucciones del 6502. El asterisco en la línea 40 de nuestro programa es una directiva, y la orden ".END" en la línea 120 es un "pseudo op". La directiva "*" se utiliza para decirle a su computador en qué lugar de la memoria se almacenará un programa después de haber sido ensamblado. La directiva ".END" se usa para decirle al ensamblador dónde debe dejar de ensamblar, y para finalizar un programa en lenguaje ensamblador.

Operandos

El campo "operando" en un programa de MAC/65 o Atari Assembler comienza a lo menos un espacio (o tabulación) después de un "op code". Los operandos se utilizan para ampliar los "op codes", haciendo que éstos sean instrucciones completas. Algunos "op codes", como CLC, CLD y RTS, no requieren operandos. Otros, como LDA, STA y ADC, requieren operandos. Vamos a entregar mucha más información acerca de los operandos más adelante.

Comentarios

Los comentarios en los programas en lenguaje ensamblador son como los comentarios en los programas BASIC: no tienen ningún efecto sobre el programa, sino que sirven para explicar los procedimientos de programación y proveer espacio para el descanso de los ojos en los listados de los programas. Hay dos maneras de escribir comentarios en los listados de código fuente escrito en el MAC/65 y el Atari Assembler Editor. Un método consiste en poner los comentarios en el campo de las etiquetas de un listado, precedido por punto y coma. El otro método consiste en ponerlos en el campo especial "Comentarios" que queda en cada línea de código a continuación de los campos de instrucción (los campos donde van los "op codes" y los operandos). Si utiliza el campo "Comentarios" al final de una línea y no tienen espacio allí para el comentario que desea escribir puede continuar sus comentarios en la siguiente línea simplemente digitando un espacio, un punto y coma, y el resto de sus comentarios.

Examinando el programa

Ahora que hemos visto nuestro programa de ejemplo campo por campo, vamos a examinarlo esta vez línea por línea.

Líneas 10 a 30 (Comentarios)

Las líneas de la 10 a la 30 corresponden a los comentarios. La línea 20 explica lo que hace el programa y las líneas 10 y 30 destacan con espacios en blanco las líneas explicativas. Es una buena práctica de programación el utilizar de manera abundante los comentarios en lenguaje ensamblador, así como en la mayoría de los otros lenguajes de programación, por lo que hemos utilizado varios comentarios en los programas de este libro.

Línea 40 (Directiva "*=")

Esta es la línea de *origen* de nuestro programa de ejemplo. Cada programa en lenguaje ensamblador debe empezar con una línea de origen. Como puede recordar del capítulo 1, lo primero que hace un computador cuando se ejecuta un programa en lenguaje de máquina es ir a una ubicación de memoria predeterminada y ver lo que se encuentra allí.

Así que cuando usted escribe un programa en lenguaje ensamblador, lo primero que tiene que hacer es decirle a su equipo dónde en su memoria tiene que empezar a buscar su programa. Cuando el ensamblador encuentra una directiva de origen, asigna la dirección que figura en la directiva al contador de programa ("program counter") del procesador 6502 de su computadora. La primera instrucción del programa será cargada en esa posición de memoria, y el resto del programa seguirá la secuencia.

La directiva de origen parece una línea sencilla de escribir, pero decidir qué número se debe poner en esta línea puede ser un trabajo muy difícil, especialmente para los programadores principiantes en lenguaje ensamblador. Hay muchos bloques de memoria en su computador que no se pueden utilizar para programas en lenguaje ensamblador, porque están reservados para otros usos (por ejemplo, para almacenar el sistema operativo de su computador, el sistema operativo de disco, el intérprete BASIC, etc.). Incluso el ensamblador que usará para escribir este programa ocupa un bloque de espacio de memoria que es territorio prohibido para usted, al menos hasta que se familiarice con ciertos puntos de referencia que puede utilizar para encontrar su camino a través de la selva de la memoria de su computador.

Mientras tanto, sucede que existe un pequeño bloque de memoria que está reservado, en circunstancias normales, sólo para el tipo de programas cortos en lenguaje ensamblador escritos por el usuario, como los que vamos a estar usando en los siguientes capítulos. El bloque de memoria se llama "Página 6", ya que va desde la dirección de memoria $0600 a $06FF (1536 a 1791 en decimal). "Página 6" tiene sólo 256 bytes de longitud, pero ese espacio es más que suficiente para el programa que vamos a ver en este capítulo, y para los otros programas que usaremos en los próximos capítulos. Más adelante, como sus programas se harán cada vez más largos, aprenderá a pasar a bloques de memoria más grandes. La línea 40 en nuestro programa de sumas le dice a su computador que el programa que va a escribir comenzará en la dirección de memoria $0600 (1536 en decimal).

Línea 50 (Línea en blanco)

Esta línea de "Comentario" sólo sirve para separar la línea de origen de las otras líneas de nuestro programa. El espacio en blanco se ve bien, ¿no?

Línea 60 (Etiqueta: ADDNRS, Mnemotécnico: "Desactivar modo Decimal")
ADDNRS CLD

ADDNRS: Hemos utilizado el campo Etiqueta en esta línea para llamar a nuestro programa ADDNRS. Así que si decide usar nuestro programa como una rutina o una subrutina en un programa más grande, ya tendrá un nombre. Entonces, podremos direccionarla por su nombre, si lo deseamos, en lugar de su ubicación en memoria. Por lo general, es una buena práctica de programación el ponerle etiquetas a las rutinas y subrutinas importantes. Una etiqueta no sólo hace una rutina más fácil de localizar y

utilizar, sino que también sirve como un recordatorio de lo que hace la rutina (o, hasta que el programa sea depurado, lo que se supone que tiene que hacer).

CLD: En este programa usamos números binarios simples, y no números en código binario decimal (BCD). Así que en esta línea vamos a desactivar la bandera de modo decimal del registro de estado del procesador 6502. La bandera decimal no necesita ser desactivada antes de cada operación aritmética en un programa, pero es una buena idea desactivarla antes de la primera suma o resta de un programa, ya que pueden haber sido activados en el transcurso de un programa anterior.

Línea 70 ("Desactivar la bandera de Acarreo")
CLC

La Bandera de Acarreo del Registro de Estado se ve afectada por tantos tipos de operaciones que se considera una buena práctica de programación el limpiar (desactivar) esta bandera antes de cada suma. Requiere sólo la mitad de una millonésima de segundo, y sólo un byte de memoria RAM. En comparación con el tiempo y la energía que la depuración nos pueda costar, es una ganga.

Línea 80 (“Cargar el Acumulador con el número 2")
LDA #2

Esta es una instrucción muy sencilla. El primer paso en una suma siempre es cargar el acumulador con uno de los números que se va a sumar. El símbolo "#" delante del número 2 significa que es un número literal, no una dirección. Si la instrucción fuese "LDA 2", entonces el acumulador se cargaría con el contenido de la dirección de memoria 0002, y no el número 2.

Línea 90 ("Sume, con acarreo, el numero 2 al contenido del Acumulador")
ADC #2

Esta es también una instrucción directa. "ADC #2" significa que el número literal 2 se añadirá al número que está en el Acumulador (En este caso, otro 2). Como hemos mencionado, no hay una instrucción en lenguaje ensamblador del 6502 que signifique "sumar sin acarreo". Así que la única manera de que una adición se pueda realizar sin acarreo es limpiando la Bandera de Acarreo del Registro de Estado y luego realizando una "suma con acarreo".

Línea 100 ("Almacene el contenido del Acumulador en la dirección de memoria $CB")
STA $CB

Esta línea completa nuestra suma. Almacena el contenido del Acumulador en la dirección de memoria $CB (decimal 203). Tenga en cuenta que el símbolo "#" no se utiliza antes del

operando (CB) en esta instrucción, ya que el operando en este caso es una dirección de memoria, y no un número literal.

Línea 110 ("Retorno desde la subrutina")
RTS

Si el mnemotécnico RTS se utiliza al final de una subrutina, funciona de la misma manera que la instrucción RETURN en BASIC: termina la subrutina y vuelve al cuerpo principal de un programa, comenzando por la línea siguiente a la línea en que aparece la instrucción RTS. Pero si se utiliza RTS al final de la parte principal de un programa, tal como en nuestro ejemplo, tiene una función diferente. En este caso, entonces, en lugar de pasar el control del programa a una línea diferente, termina todo el programa y devuelve el control del computador al dispositivo de entrada que estaba en control antes de que comenzara el programa (Por lo general un cartucho, un sistema operativo, un editor de pantalla de teclado, o un monitor de lenguaje de máquina).

Línea 120
Directiva .END

Así como la Directiva "*=" comienza un programa de lenguaje ensamblador, la Directiva ".END" le pone fin al programa. La Directiva .END indica al ensamblador detener el ensamblaje, y eso es exactamente lo que el ensamblador hace, incluso si hay más código fuente después de la directiva .END. La Directiva .END por lo tanto puede ser utilizada como una poderosa herramienta de depuración. Usted puede ponerla donde desee en un programa que se está depurando, y ahí es donde el ensamblador siempre detendrá el ensamblado, hasta que usted elimine la Directiva ".END". Cuando haya terminado de depurar su programa puede utilizar la Directiva .END para que la rutina llegue ordenadamente a su fin. Antes de que pueda hacer eso, por supuesto, deberá quitar todas las directivas .END que aún queden por ahí, si es que quiere obtener su programa final completamente ensamblado. Cuando la depuración está completa y el programa está terminado, debe contener sólo una directiva .END, y ésta debe ir donde pertenece: al final de su programa.

Ensamblado de un Programa en Lenguaje Ensamblador

OK. ¿Está listo para escribir y ejecutar su primer programa en lenguaje ensamblador? Bien. Entonces, siéntese frente a su computadora, encienda su unidad de disco y el monitor de video, y ejecute su ensamblador (si está en disco), o inserte el cartucho de ensamblador (si este es el tipo de ensamblador que tiene) en la ranura de cartuchos en su Atari. Si estás usando el MAC/65 o un cartucho Atari Assembler Editor, su ensamblador estará listo para ser usado cuando la palabra "EDIT" aparezca en su monitor de video. Este mensaje es el equivalente en lenguaje ensamblador al "READY" del BASIC. Si esto no sucede, entonces compruebe todas las conexiones de los componentes del sistema y

repita la puesta en marcha. Mientras no obtenga el mensaje "EDIT" del sistema, no se puede programar nada en lenguaje ensamblador.

Cuando tenga el ensamblador en marcha puede poner un disco vacío y formateado en su unidad de disco. Este disco debe tener un conjunto de archivos de DOS grabados en él para poder arrancar automáticamente sin necesidad de un disco de inicio al encender su computador. Si graba todos los programas de este libro en su disco de datos, aun así tendrá espacio para los archivos de DOS, los que le ahorrarán tiempo.

Digitando el Programa

Cuando el mensaje "EDIT" aparece en su pantalla, puede comenzar a digitar en su computador el programa de sumas (la versión código fuente) que viene al comienzo de este capítulo. A medida que digite el programa, tenga mucho cuidado con el espaciado que utiliza. MAC/65 y el cartucho Atari Assembler Editor son un poco quisquillosos con respecto al espaciado. En las líneas que tienen un punto y coma, recuerde que sólo debe haber un espacio entre el número de línea y el punto y coma. En la línea 40, sin embargo, debe haber al menos dos espacios entre el número de línea y el asterisco, ya que "*" es una directiva, y las directivas aparecen en el campo "op code" de los programas de MAC/65 y de Atari Assembler Editor.

En la línea 60, debe haber un espacio entre el número de línea y la etiqueta "ADDNRS", y un espacio entre "ADDNRS" y el mnemotécnico "CLD". En las líneas de la 70 a la 110 debe haber a lo menos dos espacios entre cada número de línea y el "op code" que viene a continuación. Y en la línea 120 debe haber al menos dos espacios entre el número de línea y la directiva ".END". Si comete un error al escribir una línea, puede retroceder y corregir, usando las teclas de control del cursor (flechas) de su teclado.

Listando su programa

Muy bien. Después de que haya digitado el listado del Programa 1 en su computador, escriba la palabra LIST. Debería ver en la pantalla algo así como esto:

```
EDIT
LIST

10 ;
20 ;ADDNRS.SRC
30 ;
40          *=$0600
50 ;
60 ADDNRS CLD
70          CLC
80          LDA #2
```

```
90        ADC #2
0100      STA $CB
0110      RTS
0120      .END

EDIT
```

Ahora, si tiene una impresora, puede imprimir su programa en papel. Sólo tiene que escribir:

```
LIST #P:
```

Es bastante fácil, ¿verdad? ¡Ni bien lo escribe y ya está terminado! Ahora, si su listado se ve bien, puede guardar su programa en un disco. Sólo asegúrese de que un disco formateado (de preferencia uno virgen) está en nuestra unidad de disco y la luz de preparado ("ready") está encendida.
A continuación, digite:

```
LIST #D:ADDNRS.SRC
```

Ahora debe encenderse la luz roja de arriba en la unidad de disco, y el disco en el que está guardando su programa debe empezar a girar. Cuando la luz ocupado ("busy") de su unidad de disco se apague, su código fuente habrá sido grabado en el disco con el nombre de archivo #D:ADDNRS.SRC. Le recomendamos que guarde su código fuente ADDNRS.SRC en un lugar seguro, ya que trabajaremos con este programa en capítulos posteriores. Ahora, de hecho, puede usar ese mismo código fuente para ensamblar su programa. Para hacer esto, ponga su disco con el código fuente en su unidad de disco y su cartucho Assembler Editor en su computador, y escriba el comando ASM. Tan pronto como haya hecho esto, el computador presentará en pantalla lo siguiente:

```
EDIT
ASM
PAGE 1
                 10 ;
                 20 ;ADDNRS.SRC
                 30 ;
0000             40        *=$0600
                 50 ;
0600  D8         60 ADDNRS  CLD
0601  18         70        CLC
0602  A902       80        LDA #2
0604  6902       90        ADC #2
0606  85CB       0100      STA $CB
0608  60         0110      RTS
0609             0120      .END
```

```
*** ASSEMBLY ERRORS: 0     23202 BYTES FREE(1)
PAGE 2
SYMBOLS

0600 ADDNRS

EDIT
```

(1)Esto depende de la versión de DOS que usted tenga.

```
ASM
                10 ;
                20 ; ADDNRS.SRC
                30 ;
0000            40           *=    $0600
                50 ;
0600 D8         60 ADDNRS    CLD
0601 18         70           CLC
0602 A902       80           LDA   #2
0604 6902       90           ADC   #2
0606 85CB       0100         STA   $CB
0608 60         0110         RTS
0609            0120         .END

EDIT
```

Captura de pantalla de ADDNRS.SRC ensamblado
con el Atari Assembler Editor

Si ha cometido algún error al digitar su programa, aquí es donde probablemente se enterará. Si el ensamblador encuentra un error en una línea, sonará un pitido y mostrará un mensaje de error. Es posible que no sea capaz de detectar todos los errores que usted cometa, pero cuando detecte uno, imprimirá un número de error en la pantalla (de manera similar a como lo hace el intérprete BASIC). Puede averiguar lo que significa ese número consultando el manual de usuario del MAC/65 o del Atari Assembler Editor. Si el ensamblador encuentra algún error en su programa puede digitar LIST, volver atrás y corregir. Entonces puede volver a escribir ASM e intentar una vez más ensamblar su programa. ¡Una vez que el listado de su código objeto se haya desplegado en la pantalla sin ningún mensaje de error, sabrá que su programa ha sido ensamblado correctamente y que acaba de escribir y ensamblar su primer programa en lenguaje ensamblador!

Una diferencia entre ensambladores

Hemos llegado a una diferencia importante entre el cartucho Atari Assembler Editor y el ensamblador MAC/65. Cuando ensamblamos el código fuente del programa usando el cartucho Atari Assembler Editor, el código objeto generado por el proceso de ensamblaje se almacena automáticamente en la memoria de su computador. Sin embargo, cuando ensamblamos un programa mediante el ensamblador MAC/65, el código objeto no se

almacena automáticamente en la memoria a menos que utilice una directiva especial. Esta directiva es ".OPT" (por "OPTion (opción)"). La directiva de opción se utiliza de la siguiente manera:

```
05 .OPT OBJ
```

Si tiene un ensamblador MAC/65, puede insertar esta línea en su programa ADDNRS.SRC, y el programa se almacenará automáticamente en la memoria RAM a medida que sea ensamblado.

¿Qué viene a continuación?

Una vez que haya almacenado un programa en la memoria RAM, puede hacer lo que quiera con él, como por ejemplo: ejecutarlo, imprimirlo en papel, guardarlo en disco, o guardar en la memoria de su computador. Sin embargo, antes de hacer cualquiera de estas cosas, es una buena idea echar una mirada más de cerca al programa ADDNRS, al listado de código objeto del programa en su forma final ensamblada. En la columna 1 del listado de código objeto, verá las direcciones de memoria en las que su suma se almacenará luego de haber sido cargada en la memoria de su computador. La columna 2 es el listado de código objeto real de su programa. Es esta columna la que le mostrará, en notación hexadecimal, la versión real en código máquina de su programa. Este es el significado de los números en la columna 2:

CODIGO FUENTE	CODIGO MAQUINA	SIGNIFICADO
CLD	D8	Desactivar la bandera modo decimal del registro de estado
CLC	18	Desactivar la bandera de acarreo del registro de estado
LDA #2	A9 02	Carga el Acumulador con el número 2
ADC #2	69 02	Sumar 2, con acarreo
STA $CB	85 CB	Almacenar el resultado en la dirección de memoria CB (203 en decimal)
RTS	60	Retorno de subrutina

Ahora bien, si lo desea, puede imprimir su listado ensamblado en papel. Simplemente digite ASM, #P: y obtendrá un listado impreso de su programa ensamblado.

Guardando su programa

Ahora usted ha escrito, ensamblado e impreso su primer programa en lenguaje ensamblador. Y eso significa que estamos casi listos para poner fin a esta sesión de programación. Pero antes de apagar el equipo, no sería una mala idea guardar el código objeto de nuestro programa en un disco. Así que eso haremos ahora.

¿"LIST" o "SAVE"?

Unos párrafos más atrás usted guardó el código fuente de su programa de sumas con el comando "LIST". Pero para guardar el código objeto de un programa en lenguaje ensamblador, hay dos comandos. Uno de ellos es SAVE. El otro es BSAVE. Si está usando el ensamblador MAC/65, el comando que se debe usar para guardar la versión código objeto del programa ADDNRS es BSAVE. Sin embargo, si está utilizando el cartucho Atari Assembler Editor, el comando a utilizar es SAVE.

Guardando el código objeto de un programa

Sin embargo, los comandos SAVE y BSAVE se utilizan de la misma forma. He aquí cómo se hace: Primero, escriba el comando ASM, y su computador ensamblará el código fuente de su programa de sumas. (Si ya ha hecho esto antes, no hay problema en hacerlo de nuevo). A continuación, mire la columna 1 del listado de código objeto en la pantalla y observe las direcciones de memoria en las que se ha ensamblado su programa. El programa debe comenzar en la dirección de memoria $0600 y debe terminar en la dirección de memoria $0609. Así que esta es la línea que se debe digitar si usted tiene un ensamblador MAC/65.

```
BSAVE #D:ADDNRS.OBJ<0600,0609
```

Y esta es la línea que se debe digitar si usted tiene un cartucho Atari Assembler Editor:

```
SAVE #D:ADDNRS.OBJ<0600,0609
```

Tan pronto como digite esa línea y presione la tecla RETURN, la luz ocupado (“busy”) en su unidad de disco debería encenderse, y su disco debería comenzar a girar. Cuando la luz ocupado (“busy”) se apague y su unidad de disco se detenga, el listado de código objeto de nuestro programa de sumas habrá sido ser almacenado de forma segura en el disco con el nombre de archivo #D:ADDNRS.OBJ.

¿Funcionó?

Ahora vamos a comprobar si todo esto de hacer LIST y SAVE funcionó. Primero digite el comando NEW y luego presione la tecla RETURN para borrar la memoria de su computador. A continuación, escriba "ENTER #D:ADDNRS.SRC". Su unidad de disco debe comenzar a girar, y el mensaje EDIT debe aparecer en su pantalla. Ahora, si escribe el comando LIST, el listado del código fuente del programa ADDNRS debería aparecer en nuestra pantalla de video. Ahora, si el ensamblador es un MAC/65, escriba la línea "BLOAD #D:ADDNRS.OBJ", y presione la tecla RETURN. Si tienen un cartucho Atari Assembler Editor, digite "LOAD #D:ADDNRS.OBJ". Su unidad de disco ahora debería cargar el código objeto del programa ADDNRS en el equipo, y la palabra EDIT debería

aparecer una vez más. Entonces, si digita el comando ASM de nuevo, el listado en código ensamblador del programa ADDNRS debería aparecer en su pantalla.

Avanzando

Usted ha logrado mucho en este capítulo. Ha escrito y ensamblado su primer programa en lenguaje ensamblador y, con suerte, tendrá una comprensión bastante buena de cómo esto se hace. Ha guardado su listado de código fuente y su listado de código ensamblador en un disco. Si tiene una impresora, también habrá impreso listados en papel. Y ahora, en el capítulo 5, vas a aprender a ejecutar un programa en lenguaje ensamblador.

Capítulo Cinco
Ejecución de un Programa en Lenguaje Ensamblador

Existen varias formas de ejecutar un programa en lenguaje de máquina en un computador Atari. Por ejemplo:

- Usando un comando especial de depuración (el comando "G") incluido tanto en el Ensamblador MAC/65 como en el cartucho Atari Assembler Editor.

- Ejecutando el programa por medio del sistema operativo de disco de Atari (DOS) o el sistema operativo OS/A+ (Si tiene el ensamblador MAC/65).

- Usando el utilitario AUTORUN.SYS del Atari DOS (o el programa STARTUP.EXC si es que está utilizando el sistema operativo OS/A+).

- Invocando su programa en lenguaje de máquina desde un programa BASIC.

En este capítulo, vamos a cubrir los tres primeros métodos de ejecución de programas en lenguaje de máquina. La cuarta técnica, invocar al programa en lenguaje de máquina desde un programa BASIC, será tratada en el Capítulo 8. Primero vamos a discutir la técnica para ejecutar un programa con el comando "G" que ofrecen tanto el ensamblador MAC/65 como el cartucho Atari Assembler Editor.

El monitor incluido en su ensamblador

Para utilizar el comando "G", necesitará de la ayuda de una herramienta práctica que viene gratis en el ensamblador MAC/65 y en el cartucho Atari Assembler Editor. A esta herramienta se le denomina utilitario de depuración. Si acaba de terminar de leer el capítulo 4 y todavía tiene su computador encendido, puede empezar a usar la herramienta de depuración de su ensamblador en sólo unos instantes, tan pronto como los lectores que apagan sus computadores entre cada capítulo tengan la oportunidad de prenderlos nuevamente.

Si apagó su computador al final del capítulo 4, por favor, enciéndalo nuevamente. Necesitará su disco de datos y su ensamblador funcionando. Cuando el mensaje EDIT aparezca en la pantalla, cargue en la memoria de su computador el código fuente del programa que escribió en el capítulo 4. Primero escriba la palabra "NEW", un buen hábito para cuando se quiera cargar un programa, por si acaso hay otro programa en la memoria. Luego, escriba

```
ENTER #D:ADDNRS.SRC
```

- Tal como lo hizo cuando cargó el programa ADDNRS.SRC al final del Capítulo 4. Cuando su unidad de disco deje de girar, puede comprobar si el programa se ha cargado correctamente simplemente escribiendo el siguiente comando:

```
LIST
```

Entonces debería poder ver el programa en su pantalla. Ahora vamos a ensamblar el programa. En realidad, si usted hizo los ejercicios del Capítulo 4, nuestro programa ADDNRS ya estará ensamblado y almacenado en su disco de datos. Pero el programa es tan corto que le tomara más tiempo cargar el código objeto desde el disco que ensamblarlo nuevamente. Así que eso haremos. Si está usando el ensamblador MAC/65, eche un vistazo al listado del código fuente de su programa y asegúrese que contenga la siguiente línea

```
05 .OPT OBJ
```

- De modo que sea cargado en la memoria del computador a medida que sea ensamblado. ¡Si está usando un cartucho Atari Assembler Editor, la línea 5 no debe estar en su programa! ¡La directiva .OPT no tiene ningún significado para el cartucho Atari Assembler Editor, y será marcada como un error!

Una vez que la línea 5 esté en su programa (o no lo esté, dependiendo del tipo de ensamblador que tenga), puede ensamblar el programa ADDNRS. Para ello, basta con teclear:

```
ASM
```

Su ensamblador le presentará el listado correspondiente al código objeto de su programa ADDNRS.

Luego puede utilizar la herramienta de depuración incluida en el MAC/65 o ensamblador Atari para depurar el programa y guardarlo en el disco en su forma final. Las utilidades de depuración del ensamblador MAC/65 y del Atari Assembler Editor son bastante similares. Pero tienen algunas diferencias.

Utilizando el depurador del MAC/65

El depurador integrado en el ensamblador MAC/65 se llama BUG/65. Para usarlo primero debe asegurarse de que su código fuente ADDNRS ha sido ensamblado correctamente usando la directiva ".OPT OBJ". Luego, mientras el ensamblador se encuentra en el modo EDIT, escriba el comando "CP". Esto le llevará de vuelta al sistema operativo OS/A+ de su ensamblador. Ahora, en respuesta al mensaje "D1" del sistema operativo

OS/A+, teclee "BUG65". La unidad de disco debe empezar a girar, y cuando se detenga, la pantalla con borde amarillo del BUG/65 deberá aparecer en el monitor de su computador.

Utilizando el depurador del ensamblador de Atari

Si está utilizando un cartucho Atari Assembler Editor, el poner su ensamblador en modo "DEBUG (depuración)" es aún más fácil. Sólo tiene que teclear

```
BUG
```



Entrando al Modo de Depuración

- (y no DEBUG), seguido por un retorno de carro. Esto le mostrará el siguiente mensaje en la pantalla:

```
DEBUG
```



Modo de depuración del Atari Assembler Editor

- y cuando este comando aparezca en su pantalla, podrá depurar sus programas en lenguaje ensamblador usando una serie completa de comandos.

En este capítulo trataremos sólo algunas de las muchas capacidades del paquete de depuración de su ensamblador. El depurador de su ensamblador es un paquete de software muy especial. Con él, puede hacer PEEK a los registros de memoria de su computador, y desplegar de diferentes formas el contenido de estos registros. Incluso puede ejecutar sus programas utilizando el depurador de su ensamblador, el cual puede advertirle de la presencia de muchos tipos de errores de programación mientras el programa se está ejecutando o incluso después de que éste se haya ejecutado.

Uso del paquete de depuración

Ahora vamos a mostrar cómo el monitor integrado de su ensamblador puede ayudarle a examinar el contenido de la memoria RAM de su Atari. Luego tendrá la oportunidad de ejecutar un programa en lenguaje de máquina usando el monitor que viene integrado en su ensamblador.

Desplegando el contenido de las Posiciones de Memoria

Como hemos señalado anteriormente, todas las características del monitor que viene integrado en su ensamblador pueden ser invocadas cuando éste se encuentra en modo DEBUG (depuración). Por ejemplo, cuando está en modo de depuración, puede echar un vistazo al contenido de las posiciones de memoria, usando la instrucción "D", que significa "mostrar memoria (Display memory)". El comando "D" es similar al comando PEEK de BASIC. Mediante el uso del comando "D" puede echar un vistazo a los registros de memoria de su Atari y ver su contenido. Para utilizar el comando "D", solo tiene que indicarle a su ensamblador las posiciones de memoria que desea mirar. Si escribe una "D" seguido de una dirección de memoria (expresado en número hexadecimal, por supuesto), obtendrá un listado en pantalla de la posición solicitada y las siete posiciones que le siguen. Si ya ensambló el código fuente de su programa ADDNRS, puede ver ahora mismo cómo funciona el "comando D" de su monitor. Simplemente escriba

```
D600
```

- y verá en la pantalla algo así:

```
0600 D8 18 A9 02 69 02 85 CB
```

(Si tiene el ensamblador MAC/65, verá algunos caracteres más después de las letras "CB". No les preste atención, son solo caracteres que en determinadas circunstancias representan los números de esta línea, pero que no significan nada en el contexto de lo que estamos haciendo ahora). Al revisar cualquier listado de código objeto que hemos

creado en otros ejercicios, verá que el resto de la línea no es sino un listado de código de máquina correspondiente a los primeros ocho bytes de su programa ADDNRS.OBJ. También puede utilizar el "comando D" de su monitor para ver más de ocho posiciones consecutivas de la memoria de su computador. Sólo tiene que escribir dos direcciones después de la "D", utilizando el formato

```
D5000 500F
```

- en el caso del ensamblador MAC/65, y el formato

```
D5000, 500F
```

- en el caso del cartucho Atari Assembler Editor.

Entonces el depurador de su ensamblador le mostrará un listado con el contenido de todos los registros que van desde la primera dirección hasta segunda dirección. Para ver cómo funciona el comando "D" cuando se utiliza el segundo parámetro opcional, escriba:

```
D0600 0608
```

(o D0600,0608 si tiene el ensamblador Atari)

Obtendrá un listado como el que sigue (con algunos símbolos extra añadidos si su ensamblador es un MAC/65):

```
0600 D8 19 A9 69 02 85 CB
0608 60 00
```

Esto corresponde, por supuesto, al listado desensamblado de su programa de sumas, completo, hasta el último mnemónico (la instrucción RTS).

El comando 'L' del Atari Assembler Editor

El Atari Assembler Editor también incluye el comando "L" (Listar memoria con desensamblado - List Memory with Disassembly). (El ensamblador MAC/65 también tiene un comando "L", pero es totalmente diferente. El comando "L" del MAC se utiliza para buscar cadenas de caracteres hexadecimales). Pero el comando "L" del ensamblador de Atari puede ser utilizado para mostrar listados desensamblados de programas en lenguaje de máquina. El comando "L" del Atari Assembler Editor es similar al comando "D", pero existen algunas diferencias. El comando "L" de Atari, tal como el comando "D", puede ser ejecutado usando una o dos direcciones. Cuando se utiliza solo una dirección, el comando "L" desplegará un listado con el contenido de 20 posiciones de memoria consecutivas en su computador (no el contenido de sólo ocho posiciones, como lo hace el comando "D").

Sin importar si se usa o no la segunda dirección opcional, el comando "L" desensamblará el código de máquina en las direcciones que se muestran por pantalla; al lado de cada número hexadecimal aparecerá la instrucción en lenguaje ensamblador a la que equivale el número, si es que existe. Para ver cómo funciona el comando "L", escriba lo siguiente en su ensamblador de Atari.

```
L0600,0608
```

Si intenta ingresar esa línea en el ensamblador MAC/65, solo obtendrá un pitido enojado y un mensaje que dice "COMMAND ERROR! (¡Error de comando!)". Pero si tiene un ensamblador Atari, esto es lo que verá:

```
0600    D8       CLD
0601    18       CLC
0602    A9 02    LDA    #$02
0604    69 02    ADC    #$02
0606    85 CB    STA    $CB
0608    60       RTS
```

Este es el listado del contenido real de las direcciones de memoria de la $0600 a la $0608 de su equipo, luego de que el código fuente ADDNRS.SRC haya sido ensamblado y almacenado en la RAM. Con sólo mirar el listado, podrá notar que su programa ha sido ensamblado y cargado en la memoria RAM correctamente, y que ahora está en la memoria RAM esperando a ser ejecutado. Hemos llegado al comando de depuración que se usa para ejecutar un programa, el comando "G" (de "Go" - Ejecutar).

Santa G

Afortunadamente, el comando "G" puede ser utilizado tanto en el ensamblador MAC/65 como en el cartucho Atari Assembler Editor. Mediante el uso del comando "G", le puede indicar a su computador que ejecute el código en lenguaje de máquina que comienza en cualquier posición de memoria que se especifique. Si tiene un cartucho de Atari Assembler Editor, verá que es muy fácil utilizar el comando "G". Cuando el ensamblador se encuentre en modo de depuración, teclee la letra G, seguido de la dirección de memoria en la que comienza el programa. Si está usando el Atari Assembler Editor, puede utilizar el comando "G" para ejecutar ahora mismo su programa ADDNRS. Sólo tiene que teclear

```
G0600
```

El programa se ejecutará.

Si tiene un ensamblador MAC/65, verá que el comando "G" es un poco más potente y que, en este caso, necesita de un parámetro adicional. Cuando utilice el comando "G" del MAC, puede (y en este caso debe) utilizar tanto una dirección de comienzo como una

dirección de término (o "punto de ruptura") de la rutina que desea ejecutar. Y cuando digite el parámetro de punto de ruptura de su programa, deberá marcarlo con el prefijo "@". Por ejemplo, he aquí cómo se usa el comando "G" para ejecutar el programa ADDNRS utilizando el depurador MAC/65:

```
G 0600 @0608
```

En el manual del usuario del MAC/65 y del BUG/65 se muestran más formas de usar del comando "G".

Sin campanas ni silbatos

Si su programa ADDNRS se ejecuta sin errores por medio del uso del comando "G" del MAC/65 o del Atari Assembler, no verá mucha acción en la pantalla de su computador. El programa hará tranquilamente su trabajo, que es sumar 2 más 2, y a continuación, tan rápido como un guiño, le devolverá el control de su computador. En este punto, verá en la pantalla los registros internos de su Atari. Si tiene el Atari Assembler Editor, la pantalla se verá así:

```
A=1C  X=00  Y=00  P=30  S=04
```

Si tiene el ensamblador MAC/65, la pantalla ser verá un poco más complicada:

```
A     X     Y     SP    NV_BDIZC         PC    INSTR
04    00    00    00    00100000         0608  RTS
```

Ambas pantallas cumplen la misma función: nos muestran que el programa terminó de ejecutarse y que ha dejado algunos valores en cinco de los registros del chip 6502: el acumulador, el registro X, el registro Y, el registro de estado del procesador, y el puntero de la pila. En la línea que muestra el depurador MAC/65 se ve la dirección que tenía el contador de programa (PC) cuando el programa llegó a su punto de interrupción. El último elemento en la línea (INSTR) nos muestra cual era la instrucción que estaba en esa dirección.

Toda esta información puede ser útil en algunas aplicaciones de depuración, ya que a veces es útil saber en qué condiciones ha quedado el chip 6502 después de que un programa se ha ejecutado. Pero esto no significa mucho para nosotros ahora, ya que nuestro programa ha finalizado su ejecución y realmente no nos importa lo que contienen los registros del 6502. Lo más importante para nosotros ahora es saber si nuestro programa hizo o no lo que tenía que hacer. La única manera de averiguarlo es mirando si el programa hizo lo que se le indicó, es decir, si sumó 2 más 2, y si almacenó el resultado del cálculo en la dirección de memoria $CB.

Mientras el ensamblador se encuentra en modo "depuración (debug)", puede hacer PEEK fácilmente en la dirección de memoria $CB y ver si se almacenó allí la suma de 2 más 2. Simplemente teclee:

```
DCB
```

Entonces debería ver una o dos líneas como estas en su pantalla (de nuevo, con algunas pequeñas y no muy significativas diferencias si nuestro ensamblador es el MAC/65):

```
00CB 04 00 00 00 00
00D0 00 79 00
```

¡Éxito! ¡El número 4, la suma de 2 más 2, está almacenado en la dirección de memoria $CB!

Guardando un Programa en Lenguaje de Máquina

En el capítulo 4, aprendió a guardar listados de código fuente y listados de código objeto de sus programas en lenguaje ensamblador. En el lenguaje ensamblador de Atari, los listados de código fuente se cargan en memoria con el comando ENTER, y se guardan en el disco utilizando el comando LIST. Si tiene el ensamblador MAC/65, también puede cargar el código fuente en la memoria usando el comando LOAD (cargar), y puede guardar en el disco sus listados de código fuente usando el comando SAVE (guardar). Cuando se usa el ensamblador MAC/65, se usa LIST y ENTER para guardar y cargar el código fuente de los programas sin "tokenizar" (abreviar), y se usa LOAD y SAVE para cargar y guardar el código fuente en forma abreviada o "tokenizada". Este es el mismo sistema utilizado en Atari BASIC para cargar y almacenar programas.

Si tiene un cartucho de Atari Assembler Editor, verá que los comandos LOAD y SAVE se utilizan con fines totalmente distintos. En los programas escritos con el Atari Assembler Editor, los comandos LOAD y SAVE están reservados para cargar y guardar código objeto, por lo que no se pueden usar para cargar y guardar listados de código fuente. Los listados de código objeto se cargan en memoria utilizando el comando LOAD (con el Atari Assembler) o BLOAD (con el MAC/65), y se guardan en disco utilizando los comandos SAVE (con el cartucho de Atari) o BSAVE (con el MAC/65). Los códigos fuente de los programas deben ser cargados en memoria y guardados en disco mientras el ensamblador está activo y en modo de edición. Pero los códigos objeto de los programas pueden ser cargados y guardados de dos maneras distintas.

Puede cargar y guardar el código objeto de los programas mientras el ensamblador esté activo y en modo de edición. También puede cargar y almacenar el código objeto de los programas usando el sistema operativo del disco OS/A+ o el de Atari. Para guardar el código objeto de un programa utilizando el menú del DOS de Atari, todo lo que tiene que hacer es seleccionar la opción de menú K, el comando "carga de binarios (Binary Load)".

Para guardar el código objeto de un programa con el sistema operativo OS/A+, el formato correcto es el siguiente:

```
[D1:]SAVE ADDNRS.OBJ 0600 0608
```

Sé que todo esto es un poco confuso, pero al menos tiene este capítulo para guiarse, y esto es mucho más de lo que había cuando yo estaba aprendiendo a programar en lenguaje ensamblador de Atari.

Escribiendo programas para que se ejecuten una vez que estos sean cargados

Puede utilizar tanto el menú de Atari DOS como el del sistema operativo OS/A+ para guardar los programas en lenguaje de máquina de manera que estos se ejecuten automáticamente en cuanto se hayan cargado en la memoria de su computador. Cuando se selecciona la opción "K" en el menú del Atari DOS, su equipo responde con este mensaje:

SAVE - GIVE FILE, START, END, INIT, RUN

Menu de Atari DOS 2.0

Si lo desea, puede responder a esta solicitud ingresando sólo dos direcciones: START y END. Pero el sistema también le permite utilizar dos direcciones más: INIT y RUN. Estas dos direcciones son parámetros opcionales. Cuando se guarda un programa sin utilizar estos parámetros, el programa guardado se cargará en memoria pero no se ejecutará después de haber sido leído del disco. Para ejecutar una programa que ha sido guardado sin usar los parámetros INIT o RUN, se debe usar un comando de ejecución especial: ya sea la opción M (RUN AT ADDRESS – Ejecutar en la dirección) del menú de DOS, o alguno de los comandos especiales (como el comando "G") que se encuentran disponibles en varios ensambladores, depuradores, y sistemas operativos.

Si lo desea, puede utilizar el DOS de Atari para guardar el código objeto de un programa de manera que se ejecute automáticamente cuando este se cargue en memoria. De hecho, si lo desea, puede incluso guardar el código objeto de un programa de manera que se ejecute tan pronto arranque ("boot") el disco en el que se encuentra almacenado. Para marcar un código objeto de un programa de manera que se ejecute automáticamente cuando se cargue en memoria, puede guardarlo desde el DOS usando el parámetro INIT, el parámetro RUN, o ambos. Los parámetros INIT y RUN hacen cosas distintas, por lo que no sorprende que sean utilizados para fines distintos.

Cuando se utiliza INIT, el programa empezará a ejecutarse en su dirección INIT *tan pronto como esta dirección sea cargada en memoria*. Cuando se utiliza el parámetro RUN, el programa empezará a ejecutarse en esa dirección, pero no hasta que el programa completo haya sido cargado en memoria. El parámetro INIT normalmente se usa para ejecutar rutinas dentro de un programa corto, mientras el programa se está cargando. Por ejemplo, convertir cadenas de texto de un tipo de código de caracteres a otro, de modo que todas las conversiones se hayan completado al momento en que el programa sea ejecutado. El parámetro RUN se utiliza para ejecutar un programa en su totalidad después de que éste se ha cargado en memoria.

Usando el Parámetro 'RUN'

Así es como se debe almacenar el programa ADDNRS.OBJ con el cartucho
Atari Assembler Editor utilizando el parámetro RUN, pero sin el parámetro INIT:

```
ADDNRS.OBJ,0600,0608,,0600
```

Fíjese en las dos comas entre el número 0608 y 0600. Esto significa que la instrucción INIT se ha dejado en blanco, y por lo tanto no se utilizará. Si se hubiera utilizado, sería el tercer número escrito, justo entre las comas. En cambio, la línea se ha escrito usando este otro formato:

```
ADDNRS.OBJ,START,END,,RUN
```

Por lo tanto, el programa se ejecutará de forma automática, a partir de la dirección $0600 (la dirección indicada en el parámetro RUN), tan pronto como se cargue completamente en la memoria. Si ha guardado el programa con este formato, y luego se carga el programa en la memoria de su computador, comenzará la carga en la dirección $0600 y terminará la carga en la dirección $0608. Luego comenzará a ejecutarse en la dirección $0600.

Utilizando el parámetro 'INIT'

El parámetro "INIT" del comando Binary Load del Atari Assembler Editor puede utilizarse solo o con el parámetro "RUN". Puede utilizar el comando "INIT" tantas veces como desee en su programa, para cada parte de éste que se desee ejecutar, a medida que sea ensamblado. El comando "RUN" sólo puede utilizarse una vez: para ejecutar todo el programa. Las instrucciones detalladas del uso de los comandos RUN e INIT se encuentran en el Manual de Referencia de su Atari Disk Operating System II. Por ahora, todo lo que necesitamos saber es que se puede guardar el código objeto de sus programas de manera que empiecen a ejecutarse después de que sean cargados usando el parámetro opcional "RUN" del comando "Binary Load".

Ejecutando programas en Lenguaje de Máquina usando el OS/A+

El OS/A+ no ofrece ningún parámetro INIT o LOAD para ejecutar programas en lenguaje de máquina, ya que no requiere ninguno de éstos. Para ejecutar un programa en lenguaje de máquina usando el OS/A+, todo lo que tiene que hacer es usar el comando "RUN": Sólo responda al mensaje "D1:" del sistema escribiendo la palabra RUN, seguido por la dirección de inicio del programa que desea ejecutar. Por ejemplo:

```
[D1:]RUN 0600
```

Entonces se ejecutará el archivo binario almacenado en esa dirección.

Escritura de Programas con auto-arranque

¿Alguna vez ha querido escribir un programa que arranque ("boot") por sí solo y a continuación se ejecute de manera automática, tan pronto como usted encienda su computador? ¡Bueno, puede hacerlo fácilmente, si sabe cómo usar el lenguaje Assembler! Todo lo que tiene que hacer es guardar el programa usando el utilitario AUTORUN.SYS que viene integrado en el DOS de Atari (o el utilitario STARTUP.EXC proporcionado por OS/A+, que vamos a discutir en un momento). Utilice cualquiera de estos utilitarios, y su programa se ejecutará automáticamente cada vez que arranque el disco en el que fue guardado. ¡De la misma manera que el software profesional!

Dos maneras de hacerlo

Hay varias maneras de utilizar el AUTORUN.SYS de Atari. Una forma es simplemente guardando su programa usando el nombre de archivo AUTORUN.SYS y el parámetro INIT, RUN o ambos. Otra manera es tomar un programa que haya guardado anteriormente como programa de ejecución automática y simplemente cambiar su nombre a AUTORUN.SYS. A continuación se explicará cómo se debe cambiar el nombre de un archivo a AUTORUN.SYS usando el cartucho Atari Assembler Editor. En primer lugar,

asegúrese de que el programa que desea convertir ya sea un programa autoejecutable. En otras palabras, asegúrese de que haya sido guardado utilizando el parámetro RUN, INIT, o ambos. Luego invoque al menú de DOS y seleccione la opción "E" - "RENAME FILE (Cambiar nombre de archivo)". A continuación verá este mensaje:

```
RENAME - GIVE OLD NAME, NEW
```

En respuesta a este mensaje del sistema, escriba:

```
ADDNRS.OBJ,AUTORUN.SYS
```

¡Eso es todo lo que tiene que hacer! A partir de ahora, cada vez que arranque del disco en el que está el Programa 1, éste se ejecutará de manera automática (Si lo desea, puede usar el monitor del cartucho Atari Assembler Editor, para asegurarse de que es verdad).

Usando el utilitario STARTUP.EXC del OS/A+

El utilitario STARTUP.EXC proporcionado por el OS/A+ es similar al AUTORUN.SYS ofrecido por DOS de Atari. Para utilizar el utilitario STARTUP.EXC, sólo arranque el OS/A+ y luego cambie el disco maestro del OS/A+ por el disco de datos en el que está almacenado el código objeto de su programa ADDNRS. Cuando el disco de datos esté en su lugar, responda al mensaje "D1:" del sistema operativo OS/A+ escribiendo lo siguiente:

```
TYPE E: D1:STARTUP.EXC
```

Esta es la línea que verá luego en su pantalla:

```
D1:TYPE E: D1:STARTUP.EXC
```

Cuando escriba esta línea, asegúrese de usar el mismo espaciado que se utilizó en los ejemplos. En particular, asegúrese de que hay un espacio entre "E:" y "D1: STARTUP.EXC". OS/A+, al igual que el ensamblador MAC y el Atari Assembler Editor, es bastante quisquilloso con el espaciado. Cuando esté seguro de que escribió la línea correctamente, presione la tecla RETURN y verá que la pantalla del computador se quedará en blanco por un momento. Cuando las luces se enciendan de nuevo, verá la pantalla en blanco con un cursor en la esquina superior izquierda. Cuando la pantalla en blanco y el cursor aparezcan, simplemente escriba la palabra "LOAD", seguido del nombre del archivo que desea convertir en un archivo de arranque ("boot") automático.

Por ejemplo:

```
LOAD ADDNRS.OBJ
```

A continuación, escriba la palabra "RUN", seguido de la dirección (en hexadecimal) en la que se encuentra la primera instrucción de su programa. Por ejemplo:

```
RUN 0600
```

En este punto, estas dos líneas serán lo único que verá en su pantalla:

```
LOAD ADDNRS.OBJ
RUN 0600
```

Ahora presione la tecla RETURN, seguido por [CONTROL] 3 (Que se digita, por supuesto, pulsando la tecla CONTROL y la tecla "3" al mismo tiempo). Cuando haya hecho esto, su disco de datos comenzará a girar. Cuando se detenga, el programa ADDNRS.OBJ habrá sido almacenado en su disco como un archivo de arranque ("boot") automático. Cuando haya terminado de crear el archivo STARTUP.EXC, puede comprobar si está realmente en el disco escribiendo el comando "DIR" para obtener el directorio de disco. Luego, si el archivo STARTUP.EXC está allí, puede comprobar si funciona apagando el computador y luego encendiéndolo. Cuando el equipo esté encendido, cargue el depurador en la memoria escribiendo el comando BUG65. Luego puede utilizar el comando "D" del depurador para comprobar los registros de memoria del $600 al $608, para ver si su programa se ha cargado, y el registro de memoria $CB, para ver si se ha ejecutado correctamente.

Invocando programas en lenguaje de máquina desde el BASIC

También puede ejecutar programas en lenguaje de máquina, llamándolos desde programas BASIC. Pero es un proceso complejo, que requiere la comprensión de algunas técnicas de programación bastante sofisticadas. Así que vamos a guardar nuestra explicación de la invocación a programas en lenguaje de máquina desde el BASIC hasta el capítulo 8, que estará completamente dedicado a la mezcla de programas en lenguaje ensamblador y programas en lenguaje BASIC.

Capítulo Seis
La dirección correcta

Hemos cubierto mucho terreno en los primeros cinco capítulos de este libro. Ahora tiene una idea bastante buena acerca de cómo funciona su computador, y lo que ocurre dentro del chip 6502 de su Atari cuando un programa se está ejecutando. Ahora sabe los principios de los sistemas numéricos binarios y hexadecimales, y ya sabe cómo escribir, depurar, cargar y guardar sus programas en lenguaje ensamblador. Pero en realidad solo hemos comenzado a explorar las capacidades del lenguaje ensamblador 6502.

El procesador 6502 de su computador Atari es un dispositivo increíblemente versátil. Tiene sólo siete registros, y entiende sólo 56 instrucciones. Pero aún con estas limitadas posibilidades, puede hacer cosas asombrosas. Una de las razones por las que el chip 6502 es tan versátil se debe a que puede acceder a las ubicaciones de memoria en un computador en 13 maneras diferentes. En otras palabras, el procesador 6502 cuenta con 13 diferentes modos de direccionamiento. En el mundo del lenguaje ensamblador, el modo de direccionamiento es una técnica para localizar y utilizar la información almacenada en la memoria de un computador.

En los programas presentados hasta ahora en este libro, hemos utilizado tres modos de direccionamiento: direccionamiento implícito, direccionamiento inmediato, y direccionamiento en la página cero. En este capítulo vamos a examinar estos tres modos de direccionamiento, junto con los otros diez disponibles.

Cada instrucción del 6502 debe utilizar uno de los modos de direccionamiento. Ninguna instrucción es capaz de utilizar todos los modos de direccionamiento, y no hay un modo de direccionamiento que sea utilizado por todas las instrucciones. La instrucción le indica al procesador qué hacer y el modo de direccionamiento le dice al proceso con qué hacerlo. En la página siguiente encontrará la lista completa de modos de direccionamiento y su formato de operación, para que pueda distinguir cuál modo está utilizando. Estos formatos son estándares para el microchip 6502 de modo que deben ser entendidos por la mayoría de los ensambladores del 6502.

Los Modos de Direccionamiento del 6502

Los 13 modos de direccionamiento del procesador 6502 son:

	MODO DE DIRECCIONAMIENTO	FORMATO
1.	Implícito (Implicado)*	RTS
2.	Acumulador	ASL A
3.	Inmediato*	LDA #2
4.	Absoluto	LDA $5000

5.	Página Cero*	STA $CB
6.	Relativo	BCC LABEL
7.	Absoluto Indexado, X	LDA $5000,X
8.	Absoluto Indexado, Y	LDA $5000,Y
9.	Página Cero, X	LDA $CB,X
10.	Página Cero, Y	STX $CB,Y
11.	Indexado Indirecto	LDA ($B0,X)
12.	Indirecto Indexado	LDA ($B0),Y
13.	Indirecto	JMP ($5000)

"ADDNRS.SRC" Revisitado

Las tres instrucciones marcadas con un asterisco * son las que hemos utilizado hasta ahora en este libro. Las tres se usaron en "ADDNRS.SRC", el programa de sumas de 8-bits introducido algunos capítulos atrás, y que ahora volvemos a ver:

Código Fuente del programa ADDNRS.SRC

```
10    ;
20    ; Programa de sumas de 8 bits
30    ;
40              *=$0600
50    ;
60    ADDNRS    CLD      ; Dirección implícita
70              CLC      ; Dirección implícita
80              LDA #02  ; Dirección inmediata
90              ADC #02  ; Dirección inmediata
100             STA $CB  ; Dirección de la página cero
110             RTS      ; Dirección implícita
```

En este ejemplo, los tres modos de direccionamiento del programa se identifican en la columna de comentarios. Ahora veamos cada uno de estos modos de direccionamiento.

Direccionamiento Implícito (o Implicado)
(Líneas 60, 70 y 110)
Formato: CLD, CLC, RTS, etc.

Cuando se utiliza el direccionamiento implícito, todo lo que tiene que escribir es una instrucción en lenguaje ensamblador de tres letras: el direccionamiento implícito no requiere el uso de un operando (de hecho no lo permite).

La instrucción en un direccionamiento implícito es similar a un verbo intransitivo en inglés: no tiene objeto. No se especifica la dirección a la que se refiere (si es que se refiere a una dirección), sino que ésta se da a entender por el mnemotécnico propiamente tal. Así que no se requiere ni se permite un operando en el direccionamiento implícito. Los mnemotécnicos de códigos de operación ("op codes") que se puede utilizar en el modo de

direccionamiento implícito son BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, y TYA.

Direccionamiento Inmediato
(Líneas 80 y 90)
Formato: LDA #02, ADC #02, etc.

Cuando se utiliza el direccionamiento inmediato en una instrucción de lenguaje ensamblador, el operando que viene a continuación del código de operación ("op code") es un número literal, y no la dirección de una posición de memoria. Así, en una orden que utiliza direccionamiento inmediato, el símbolo "#" de un número literal siempre aparece delante del operando. Cuando se utiliza una dirección inmediata en una instrucción en lenguaje ensamblador, el ensamblador no tiene que ir y mirar en la posición de memoria para encontrar el valor. En cambio, el valor es asignado directamente en el acumulador. Así, cualquier operación que sea llamada por la orden podrá ser ejecutada de inmediato. Las instrucciones que pueden ser usadas en modo de direccionamiento inmediato son ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, y SBC.

Direccionamiento de Página Cero
(Línea 100)
Formato: STA $CB, etc.

No es difícil distinguir entre una orden que utiliza direccionamiento inmediato y una que utiliza el direccionamiento de página cero. En una orden que utiliza el direccionamiento de página cero, el operando siempre consta de un solo byte, el que puede ir desde $00 hasta $FF. Y este número equivale a una dirección que está dentro de un bloque de la memoria RAM llamado "página cero".

El símbolo "#" no se utiliza en el direccionamiento de página cero, porque el operando en este tipo de direccionamiento es siempre una posición de memoria, y nunca un número literal. Así que la operación que es invocada en la orden se ejecuta sobre el contenido de la posición de memoria especificada, y no sobre el operando en sí. Las direcciones de la página cero usan operandos de un solo byte porque eso es todo lo que necesitan. Como acabamos de decir, las posiciones de memoria a las que hacen referencia se encuentran en un bloque de memoria de su computador que se llama, lógicamente, página cero. Y para direccionar una posición de memoria en la página cero, un operando de un solo byte es todo lo que se necesita.

En concreto, el bloque de memoria de su computador al que se le conoce como página cero se extiende desde las dirección de memoria $00 hasta la dirección de memoria $FF. También se puede decir (correctamente) que la página cero se extiende desde la dirección $0000 hasta la dirección $00FF. Pero en realidad no es necesario usar los pares de ceros extra cuando se quiere hacer referencia a una dirección de la página cero. Cuando usted pone una dirección de un solo byte a continuación de una instrucción en

lenguaje ensamblador, el computador sabe que la dirección está en la página cero. Dado que las direcciones de la página cero usan operandos de un solo byte, la página cero es un área preciada en la memoria RAM de su Atari. De hecho es tan preciada, que la gente que diseñó su computador se adueñó de la mayor parte de ella. Casi toda la página cero es utilizada por el sistema operativo de su computador y otras rutinas esenciales, y no dejan mucho espacio para los programas escritos por los usuarios.

Más adelante en este libro, en un capítulo dedicado a la gestión de memoria, vamos a discutir en detalle el espacio de memoria que se encuentra disponible en la página cero. Por ahora, lo importante es recordar que la página cero es un modo de direccionamiento que utiliza una dirección de memoria en la página cero como un operando de un solo byte. Las instrucciones que se pueden utilizar con el direccionamiento de página cero son ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, y STY.

Nuevos modos de direccionamiento

Ahora vamos a describir los cinco modos de direccionamiento del 6502 que no hemos visto hasta ahora:

Direccionamiento del Acumulador
Formato: ASL A

El modo de direccionamiento del acumulador se utiliza para realizar una operación sobre un valor almacenado en el acumulador del procesador 6502. El comando ASL A, por ejemplo, se utiliza para desplazar ("shift") una posición en cada bit en el acumulador, provocando que el bit de más a la izquierda (el bit 7) sea asignado al bit de acarreo del registro de estados del procesador (P). Otras instrucciones que pueden ser utilizadas con el modo de direccionamiento del acumulador son LSR, ROL, y ROR.

Direccionamiento Absoluto
Formato: STA $5000

El direccionamiento absoluto es similar al direccionamiento de página cero. En una orden que utiliza el direccionamiento absoluto, el operando es una dirección de memoria, no un número literal. La operación invocada en una orden con direccionamiento absoluto se realiza siempre sobre el valor almacenado en la posición de memoria especificada, y no sobre el operando. La diferencia entre una dirección absoluta y una dirección de página cero es que una orden de direccionamiento absoluto no tiene que estar en la página cero, sino que puede estar en cualquier lugar de la memoria RAM libre. Por lo tanto, una orden de direccionamiento absoluto requiere un operando de dos bytes, y no un operando de un byte, que es todo lo que necesita una dirección de la página cero.

Así se ve nuestro programa ADDNRS.SRC si se usa direccionamiento absoluto en lugar de direccionamiento de página cero:

Código fuente del programa "ADDNRS"
(Con direccionamiento absoluto en la línea 100)

```
10    ;
20    ; Programa de sumas de 8 bits
30    ;
40            *=$0600
50    ;
60    ADDNRS  CLD         ; Dirección implícita
70            CLC         ; Dirección implícita
80            LDA #02     ; Dirección inmediata
90            ADC #02     ; Dirección inmediata
100           STA $5000   ; Dirección absoluta
110           RTS         ; Dirección implícita
```

El único cambio que se ha hecho a este programa está en la línea 100. El operando en esa línea ahora es un operando de dos bytes, y el cambio hace que el programa sea un byte más largo. Pero ahora la dirección en la línea 100 ya no tiene que estar en la página cero. Ahora puede ser la dirección de cualquier byte en la RAM libre.

Mnemotécnicos que pueden utilizar el modo de direccionamiento absoluto son ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, y STY.

Direccionamiento Relativo
Formato: BCC NEXT

El direccionamiento relativo es un modo de direccionamiento utilizado por una técnica llamada ramificación ("branching") o salto condicional, un método para ordenarle a un programa que salte a una rutina determinada bajo ciertas condiciones en el lenguaje ensamblador del 6502. Hay ocho instrucciones de salto condicional, o mnemotécnicos de dirección relativa. Los ocho comienzan con la letra "B", que significa "branch to - saltar a". A continuación se presentan algunos ejemplos de instrucciones de salto condicional que utilizan el direccionamiento relativo:

BCC (Salta a la dirección especificada, si la bandera de acarreo está desactivada.)
BCS (Salta a la dirección especificada, si la bandera de acarreo está activada).
BEQ (Salta a la dirección especificada, si la bandera Cero está activada).
BNE (Salta a la dirección especificada, si la bandera Cero está desactivada.)

Las ocho instrucciones de salto condicional serán descritas más adelante en un capítulo dedicado a los bucles y saltos.

Lo que hacen las instrucciones de comparación

Los ocho mnemotécnicos de salto condicional a menudo se usan junto con otras tres instrucciones llamadas instrucciones de comparación. Normalmente, una instrucción de comparación se usa para comparar dos valores entre sí, y luego se usa una instrucción de salto condicional para determinar qué debe hacerse si la comparación resulta de una u otra manera. Las tres instrucciones de comparación son:

CMP ("comparar el número en el acumulador con...")

CPX ("comparar el valor en el registro X con...")

CPY ("comparar el valor en el registro Y con...")

Las instrucciones de salto condicional también pueden venir a continuación de operaciones aritméticas o lógicas, o de pruebas de bits y bytes. Por lo general, una instrucción de ramificación causa que un programa salte a una dirección determinada, si ciertas condiciones se cumplen o no. Un salto podría ejecutarse si, por ejemplo, un número es mayor que otro, si dos números son iguales, o si el resultado de una operación devuelve un número positivo, negativo o cero.

Un ejemplo de salto condicional

Acá tenemos un ejemplo de una rutina en lenguaje ensamblador que utiliza el salto condicional:

Programa de sumas de 8 Bits con verificación de error

```
10    ;
20    ; SUMA DE 8-BIT CON VERIFICACION DE ERROR
30    ;
40             *=$0600
50    ;
60    ADD8BTS  CLD
70             CLC
80             LDA $5000
90             ADC $5001
100            BCS ERROR
110            STA $5002
120            RTS
130   ERROR    RTS
```

Este es un programa de sumas de números de 8 bits que incluye una sencilla comprobación de errores. Se suman dos números de 8 bits usando direccionamiento absoluto. Si este cálculo da como resultado un valor de 16 bits (un número mayor que 255), habrá un error de desbordamiento, y además, el bit de acarreo del registro de estado del procesador se activará. Si no se activa el bit de acarreo, entonces la suma de

los valores almacenados en las direcciones $5000 y $5001 se guardará en la dirección $5002. Sin embargo, si se activa el bit de acarreo, esta condición será detectada en la línea 100, y el programa saltará hacia la línea con la etiqueta ERROR (es decir, la línea 130). En la línea 100, usted puede comenzar cualquier tipo de rutina que necesite: por ejemplo, puede escribir una rutina que imprima un mensaje de error en la pantalla. Sin embargo, en este programa de ejemplo, un error sólo deriva en la ejecución de la instrucción RTS.

Direccionamiento Absoluto Indexado

Formato: LDA $0500,X o LDA $0500,Y

Una dirección indexada, tal como una dirección relativa, se calcula usando un desplazamiento ("offset"). Pero en una dirección indexada, el desplazamiento está determinado por el contenido actual del registro X o del registro Y del 6502. Una orden que contiene una dirección indexada puede ser escrita utilizando cualquiera de los siguientes formatos:

```
LDA $5000,X
```
o
```
LDA $5000,Y
```

Cómo funciona el Direccionamiento Indexado Absoluto

Cuando se utiliza el direccionamiento indexado en una instrucción en lenguaje ensamblador, el contenido ya sea del registro X o del registro Y (dependiendo del registro índice que se esté utilizando) es sumado a la dirección indicada en la instrucción para determinar la dirección final. He aquí un ejemplo de una rutina que usa direccionamiento indexado. La rutina está diseñado para recorrer cada uno de los bytes de una cadena (“String”) de caracteres ATASCII (El código ASCII de Atari), guardando la cadena en un buffer de texto. Cuando la cadena se haya almacenado en el buffer, la rutina termina. El texto que se va a mover tiene la etiqueta TEXT y el buffer que va a ser llenado con el texto tiene la etiqueta TXTBUF. La dirección inicial de TXTBUF, y el código ATASCII para el retorno de carro ("RETURN") se definen en una tabla de símbolos que precede al programa.

RUTINA PARA MOVER UN BLOQUE DE TEXTO
(Un Ejemplo de Direccionamiento Indexado)
MOVEBLOC.SRC

```
10   ;
20   ; Rutina para mover un bloque de texto
30   ;
40   TXTBUF = $5000
50   EOL = $9B
60   ;
```

```
70          *=$600
80    ;
90    TEXT    .BYTE $54,$41,$4B,$45,$20,$4D,$45,$20
100           .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
110           .BYTE $4C,$45,$41,$44,$45,$52,$21,$9B
120   ;
130   DATMOV
140   ;
150           LDX #0
160   LOOP    LDA TEXT,X
170           STA TXTBUF,X
180           CMP #EOL
190           BEQ FINI
200           INX
210           JMP LOOP
220   FINI    RTS
230           .END
```

Verificando el retorno de carro

Al comenzar el programa, sabemos que la cadena termina con un retorno de carro (Código ATASCII $9B), como habitualmente terminan las cadenas en los programas Atari. A medida que avanza el programa a través de la cadena, comprueba cada carácter para ver si se trata o no de un retorno de carro. Si el carácter no es un retorno de carro, el programa pasa al siguiente carácter. Si el carácter es un retorno de carro, significa que no hay más caracteres en la cadena, y la rutina termina.

Direccionamiento Página Cero, X

Formato: LDA $CB, X

El direccionamiento "Página Cero, X" se utiliza tal como el "Direccionamiento Indexado, X". Sin embargo, la dirección utilizada en el modo de direccionamiento "Página Cero, X" se encuentra (lógicamente) en la página cero. Las instrucciones que se pueden utilizar en el modo de direccionamiento "Página Cero, X" son ADC, AND, ASL, CMP, DEC, EOR, INC, LDA LDY, LSR, ORA, ROL, ROR, SBC, STA, y STY.

Direccionamiento Página Cero, Y

Formato: STX $CB, Y

El direccionamiento "Página Cero, Y" funciona tal como el direccionamiento "Página Cero, X", pero se puede utilizar sólo con dos mnemotécnicos: LDX y STX. Si no fuera por el direccionamiento "Página Cero, Y", no sería posible utilizar el direccionamiento absoluto indexado con las instrucciones LDX y STX - esta es la única razón por la que existe este modo de direccionamiento.

Direccionamiento Indirecto

Hay dos subcategorías de direccionamiento indexado: direccionamiento indirecto indexado, y direccionamiento indexado indirecto. Ambos direccionamientos se utilizan principalmente para buscar datos almacenados en tablas. Si piensa que los nombres de estos modos de direccionamiento son confusos, no es el primero en quejarse. Nunca pude entenderlos hasta que ideé un truco para ayudarme a terminar con la confusión.

Este es el truco: El Direccionamiento Indirecto Indexado, que tiene una "x" en la última palabra de su nombre, es un modo de direccionamiento que hace uso del registro X del chip 6502. El Direccionamiento Indexado Indirecto, que no tiene una "x" en la última palabra de su nombre, utiliza el registro Y del 6502.

Ahora vamos a revisar los dos modos de direccionamiento indirecto de su Atari, partiendo por el Direccionamiento Indirecto Indexado.

Direccionamiento Indirecto Indexado
Formato: ADC($C0,X)

El Direccionamiento Indirecto Indexado trabaja en varios pasos. En primer lugar, el contenido del registro X se suma a una dirección de la página cero - no el contenido de la dirección, sino a la dirección en sí misma. El resultado de este cálculo debe ser siempre otra dirección de la página cero. Cuando esta segunda dirección se ha calculado, se toma el valor que contiene esta dirección, junto con el contenido de la siguiente dirección, y ambos constituyen una tercera dirección. Esta tercera dirección es (por fin) la dirección que finalmente será interpretada como el operando de la instrucción en cuestión.

Un ejemplo de direccionamiento indirecto indexado

Un ejemplo puede ayudar a clarificar este proceso.

Supongamos que la dirección de memoria $B0 en su computador contiene el número $00, y que la dirección de memoria $B1 contiene el número $06, y el registro X contiene el número 0. Éstas son las equivalencias correspondientes en una forma más fácil de leer:

```
$B0 = #$00
$B1 = #$06

X   = #$00
```

Ahora vamos a suponer que está ejecutando un programa que contiene la instrucción indirecta indexada LDA ($B0, X). Si todas estas condiciones existen cuando el equipo encuentre la instrucción LDA ($B0, X), el equipo sumará el contenido del registro X (un

cero) al número $B0. La suma de $B0 y 0, por supuesto, es $B0. Así que el equipo irá a las direcciones de memoria $B0 y $B1, encontrará el número $00 en la dirección de memoria $B0, y el número $06 en la dirección $B1.

Ya que los computadores basados en el 6502 almacenan los números de 16 bits en orden inverso o sea, el byte bajo primero, el computador interpretará el número que se encuentra en $B0 y $B1 como $0600. Por lo que cargará el acumulador con el número $0600, el valor de 16 bits que estaba almacenado en $B0 y $B1. Ahora imaginemos qué pasa cuando el equipo encuentra la instrucción LDA ($ B0, X), y el registro X del 6502 contiene el número 04, en lugar del número 00. Aquí tiene un gráfico que ilustra los valores, además de otras equivalencias que usaremos en breve:

```
$B0 = #$00
$B1 = #$06
$B2 = #$9B
$B3 = #$FF
$B4 = #$FC
$B5 = #$1C

X   = #$04
```

Si estas son las condiciones cuando el equipo encuentre la instrucción LDA ($B0, X), entonces el computador sumará el número $04 (el valor en el registro X) al número $B0, y después irá a las direcciones de memoria $B4 y $B5. En estas dos direcciones encontrará finalmente la dirección (primero el byte menor, por supuesto) del dato que estaba buscando, en este caso, $1CFC.

Un modo rara vez usado

El Direccionamiento Indirecto Indexado no se utiliza mucho en programas de lenguaje ensamblador. Se usa cuando se desea localizar una dirección de 16-bits que se encuentra en una tabla de direcciones almacenada en la página cero. Dado que encontrar espacio disponible en la página cero es difícil, es poco probable que pueda almacenar muchas tablas de datos en ella. Así que es muy probable que casi nunca use el direccionamiento indirecto indexado.

Direccionamiento Indexado Indirecto
Formato: ADC ($C0), Y

El Direccionamiento Indexado Indirecto no es tan escaso como el Direccionamiento Indirecto Indexado. De hecho, es utilizado en programas en lenguaje ensamblador en muchas ocasiones. El direccionamiento Indexado Indirecto utiliza el registro Y (nunca el registro X) como desplazamiento ("offset") para el cálculo de la dirección base del comienzo de una tabla. La dirección de inicio de la tabla tiene que estar almacenada en la

página cero, pero la tabla no tiene que estar ahí. Cuando un ensamblador encuentra una dirección indexada indirecta en un programa, lo primero que hace es mirar en la página cero el contenido de la dirección que viene entre paréntesis y que precede a la "Y". El valor de 16 bits almacenado en esa dirección y en la dirección que viene a continuación se suma al contenido del registro Y. El valor resultante es una dirección de 16 bits: y corresponde a la dirección que la instrucción está buscando.

Un ejemplo de direccionamiento indexado indirecto

He aquí un ejemplo de direccionamiento indexado indirecto:

Su computador está ejecutando un programa y llega a la instrucción "ADC ($ B0), Y". A continuación, examina las direcciones de memoria $B0 y $B1. En $B0 se encuentra el número $00. En $B1, se encuentra el número $50. Y el registro Y contiene el número 4. Aquí tiene un gráfico que ilustra las condiciones:

```
$B0 = #$00
$B1 = #$50

Y   = #$04
```

Si estas son las condiciones cuando el computador encuentre la instrucción "ADC ($B0), Y", entonces combinará los números $00 y $50, obteniendo (bajo la forma peculiar del 6502 o sea, el byte más bajo primero) la dirección $5000. A continuación sumará el contenido del registro Y (4 en este caso) al número $5000, y terminará obteniendo un total de $5004. Esta cifra, $5004, será el valor final del operando ($B0, Y). Por lo tanto el contenido del acumulador será sumado al número que se almacena en la dirección de memoria $5004.

Una vez que entienda el direccionamiento indexado indirecto, verá que puede convertirse en una herramienta muy valiosa en la programación en lenguaje ensamblador. Una sola dirección, la dirección de comienzo de la tabla, tiene que estar almacenada en la página cero, donde el espacio disponible siempre es escaso. Sin embargo, esa dirección, sumada al contenido del registro de Y, puede ser utilizada como un punto de partida para localizar cualquier otra dirección en la memoria de su computador. A medida que se familiarice con el lenguaje ensamblador, tendrá muchas oportunidades para ver cómo funciona el direccionamiento indirecto. Encontrará ejemplos de esta técnica en algunos programas de este libro, y encontrará muchos más ejemplos en otros programas en lenguaje ensamblador.

Direccionamiento Indirecto

Formato: JMP ($5000)

En el lenguaje ensamblador del 6502, el direccionamiento indirecto no indexado puede ser utilizado con un solo mnemotécnico: JMP. Un ejemplo de direccionamiento indirecto no indexado es la instrucción JMP ($ 5000), que significa "Saltar a la posición de memoria que se encuentra almacenada en las direcciones de memoria $ 5000 y $ 5001."

El concepto 'LIFO'

La pila es lo que los programadores a veces llaman bloque de memoria "LIFO" (Last In First Out - El primero en entrar es el primero en salir). Funciona como una pila de platos en un restaurante. Cuando usted pone un número en la posición de memoria que está en la parte superior de la pila, cubre el número que antes estaba en la cima de la pila. Así que se debe quitar el número que está en la parte superior de la pila antes de poder acceder al número que está debajo de este, el mismo que anteriormente estaba en la parte superior de la pila.

Cómo el 6502 usa la pila

El procesador 6502 a menudo usa la pila para el almacenamiento temporal de datos durante la operación de un programa. Por ejemplo, cuando un programa salta a una subrutina, el 6502 toma la dirección de memoria a la que el programa más adelante tendrá que volver, y deja ("push") esa dirección en la cima de la pila. Luego, cuando la subrutina termina con una instrucción RTS, la dirección de retorno se extrae ("pull") de la parte superior de la pila y se carga en el contador de programa del 6502. Luego, el programa puede volver a la dirección apropiada, y el procesamiento normal puede reanudarse. La pila también se utiliza a menudo en programas escritos por el usuario. He aquí un ejemplo de una rutina que hace uso de la pila. Puede reconocer que es una variación del programa de sumas de 8 bits que hemos estado utilizando.

No trate de ejecutar este programa hasta que entienda cómo funciona la pila y cómo impedir que el programa falle.

RUTINA DE SUMAS QUE USA EL STACK
STACKADD.SRC

```
10    ;
20    ; STACKADD.SRC
30    ;
40              *=$0600
50    ;
60    ; CUANDO COMIENZA ESTE PROGRAMA, DOS
70    ; NUMEROS DE 8 BITS ESTAN EN LA PILA
80    ;
90    STKADD
100          CLD
105          CLC
```

```
110          PLA
120          STA $B0
130          PLA
140          ADC $B0
150          STA $C0
160          RTS
170          .END
```

Este programa es una rutina directa y simple de sumas que muestra lo fácil y cómodo que es utilizar la pila en programas en lenguaje ensamblador. En la línea 110, se toma un valor de la pila y se almacena en el acumulador. Luego, en la línea 120, el valor se almacena en la dirección de memoria $B0. En las líneas 130 y 140, se toma otro valor de la pila, y se agrega ahora al valor almacenado en $B0. El resultado de este cálculo se almacena en $C0, y con esto finaliza la rutina. Este es sólo un pequeño ejemplo de las muchas formas en que se puede usar la pila.

Encontrará otras maneras de utilizar la pila en los últimos capítulos de este libro. Si se preocupa de manejar la pila adecuadamente, o sea, si limpia la pila después de cada uso, puede llegar a ser una herramienta de programación muy potente. ¡Pero, si enreda la pila mientras la está usando, seguro que tendrá problemas!

Los mnemotécnicos que hacen uso de la pila son:

PHA ("empujar el contenido del acumulador en la pila")

PLA ("sacar el valor superior de la pila y depositarlo en el acumulador")

PHP ("empujar el contenido del registro P en la pila")

PLP ("sacar el valor superior de la pila y depositarlo en el registro P")

JSR ("poner el PC actual en la pila y saltar a la dirección")

RTS ("sacar la dirección de retorno de la pila y ponerla en el PC e incrementarlo en uno." Esto provoca que la ejecución continúe donde quedó anteriormente.)

Las operaciones PHP y PLP a menudo se incluyen en subrutinas en lenguaje ensamblador para que el contenido del registro P no sea eliminado mientras se ejecutan las subrutinas. Al saltar a una subrutina que puede cambiar el estado del registro P, siempre es una buena idea comenzar la subrutina empujando el contenido del registro P a la pila. Luego, justo antes de terminar la subrutina, puede restaurar el estado anterior del registro de P con una instrucción PHP. De esta forma, el contenido del registro P no será destruido mientras se ejecuta la subrutina.

Capítulo Siete
Saltando y dando vueltas

Ahora vamos a divertirnos un poco con el lenguaje ensamblador de Atari. En este capítulo, aprenderá a imprimir mensajes en la pantalla, codificar y decodificar caracteres ATASCII (El código ASCII de Atari), y cómo realizar otros trucos en lenguaje ensamblador. Vamos a realizar estas hazañas usando técnicas avanzadas de programación en lenguaje ensamblador que no hemos tratado hasta ahora, junto con nuevas variantes de las técnicas que hemos visto en capítulos anteriores. Estas son algunas de las técnicas de programación que vamos a cubrir en este capítulo.

- Uso de la directiva .BYTE del lenguaje ensamblador.
- Incremento y decremento de los registros X e Y.
- Uso en conjunto de las instrucciones de comparación y salto ("branch").
- Uso avanzado de ciclos y saltos.
- Escritura de instrucciones reubicables en lenguaje ensamblador.

Sin embargo, antes de empezar, haré un truco muy astuto. Voy a pedirle que escriba y guarde en un disco un programa que no le he presentado anteriormente. Es probable que no lo entienda, a menos que tenga experiencia previa en programación en lenguaje ensamblador. Le pido que escriba este programa porque contiene un par de rutinas que son necesarias para ejecutar otros dos programas que le presentaré y explicaré más adelante en este capítulo. El programa, que puede que no entienda, será explicado en detalle en el capítulo 12, "E/S y Usted".

Dos buenas razones

En este programa encontrará dos sofisticadas pero muy útiles subrutinas. Una de ellas es una rutina que abre la pantalla como dispositivo de salida y pone su Atari en modo de edición. Ya tendrá ese trabajo hecho la próxima vez que lo encuentre, en el capítulo "E/S y Usted". Así que espero que mire hacia el futuro y vea pastos más verdes en el último capítulo de este libro, y no se enoje demasiado conmigo por pedirle que escriba ahora este programa.

PROGRAMA PARA IMPRIMIR EN LA PANTALLA

```
10    ;
20    .TITLE "RUTINA PRNTSC"
30    .PAGE "RUTINAS PARA IMPRIMIR EN LA PANTALLA"
40    ;
50             *=$5000
60    ;
70    BUFLEN=23
80    ;
```

```
0100  EOL=$9B
0105  ;
0110  OPEN=$03
0120  OWRIT=$0B
0130  PUTCHR=$0B
0135  ;
0140  IOCB2=$20
0170  ICCOM=$342
0180  ICBAL=$344
0190  ICBAH=$345
0200  ICBLL=$348
0210  ICBLH=$349
0220  ICAX1=$34A
0230  ICAX2=$34B
0235  ;
0240  CIOV=$E456
0250  ;
0260  SCRNAM     .BYTE "E:",EOL
0270  ;
0280  OSCR       LDX #IOCB2 ; RUTINA PARA ABRIR LA
0300             LDA #OPEN  ; PANTALLA
0310             STA ICCOM,X
0320  ;
0330             LDA #SCRNAM&255
0340             STA ICBAL,X
0350             LDA #SCRNAM/256
0360             STA ICBAH,X
0370             LDA #OWRIT
0380  ;
0390             STA ICAX1,X
0400             LDA #0
0410             STA ICAX2,X
0420             JSR CIOV
0430  ;
0440             LDA #PUTCHR
0450             STA ICCOM,X
0460  ;
0470             LDA #TXTBUF&255
0480             STA ICBAL,X
0490             LDA #TXTBUF/256
0500             STA ICBAH,X
0510             RTS
0520  ;
0530  PRNT
0540             LDX #IOCB2
0550             LDA #BUFLEN&255
0560             STA ICBLL,X
0570             LDA #BUFLEN/256
0580             STA ICBLH,X
0590             JSR CIOV
0600             RTS
0610  ;
```

```
0620  TXTBUF=*
0630  ;
0640             *=*+BUFLEN
0650  ;
0660  .END
```

¡Ahora guárdelo!

Cuando haya escrito este programa, puede ensamblar su código objeto y guardarlo en un disco usando el nombre PRNTSC.OBJ. A continuación encontrará otro programa que me gustaría que escriba y ensamble. Es el único con el que vamos a trabajar el resto de este capítulo.

THE VISITOR

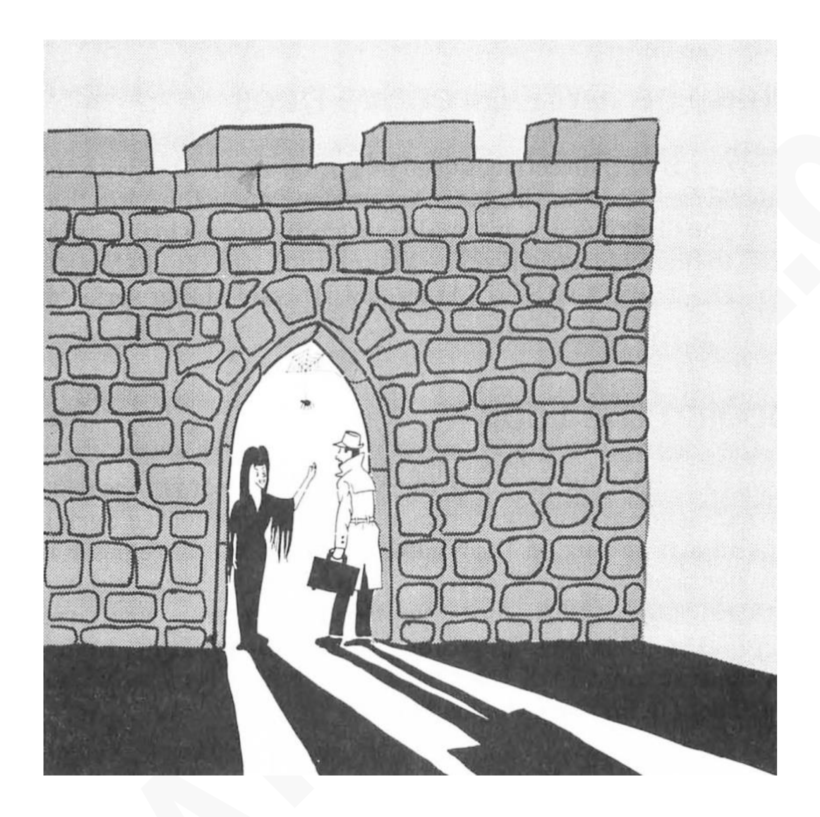
```
10      ;
20      ; EL VISITANTE
30      ;
35      TXTBUF=$5041
40      OPNSCR=$5003
50      PRNTLN=$5031
70      ;
80               *=$600
90      ;
0100    TEXT     .BYTE $4C,$4C,$45,$56,$41,$4D,$45,$20
0110             .BYTE $43,$4F,$4E,$50,$20,$54,$55,$20
0120             .BYTE $4C,$49,$44,$45,$52,$21
0130    ;
0140    VIZTOR
0150    ;
0160             LDX #0
0170    LOOP     LDA TEXT,X
0180             STA TXTBUF,X
0190             INX
0200             CPX #23
0210             BNE LOOP
0220             JSR OPNSCR
0230             JSR PRNTLN
0240    INFIN    JMP INFIN
```

Este programa se llama (por razones que pronto descubrirá) The Visitor (El Visitante). Es un programa que está diseñado para imprimir un mensaje críptico en nuestra pantalla.

Ejecutando ‘The Visitor’

Cuando haya terminado de escribir The Visitor, puede ejecutarlo de inmediato. Sólo ensámblelo, y luego cargue el código objeto del programa PRNTSC en su computador. Entonces, puede ejecutar The Visitor, ya sea poniendo el computador en modo DEBUG y escribiendo G617, o entrando al modo DOS y ejecutándolo con un comando de DOS.

Cuando haya terminado de escribir el programa, no sería una mala idea guardarlo también en un disco. El nombre de archivo sugerido para el programa es VISITOR.SRC. Luego de que haya ejecutado y guardado su programa, sabrá exactamente lo que éste hace. Así que ahora podremos explicar cómo es que hizo lo que acaba de hacer. Empezaremos con una explicación de la Directiva .BYTE del lenguaje ensamblador, que verá desde la línea 100 a la 120.



La Directiva. BYTE

A la directiva .BYTE a veces se le llama pseudo código de operación, o pseudo-op, porque aparece en la columna de op-codes del listado de código fuente en lenguaje ensamblador. Pero en realidad no forma parte del conjunto de instrucciones del lenguaje ensamblador del 6502. En cambio, es una directiva especializada diseñada para ser usada en algunos ensambladores, pero no con todos. Por ejemplo, .BYTE funciona con en el ensamblador MAC/65 y con el Atari Assembler Editor, pero no funciona con el Atari Macro Assembler and Text Editor. Al escribir un programa con el Atari Macro Assembler and Text Editor, tiene que usar las letras DB en lugar de la directiva .BYTE. Otros pseudo códigos de operación también son distintos entre ensambladores. No hay estándares generales para la escritura de directivas, por lo que los pseudo códigos de operación diseñados para un ensamblador a menudo no funcionan en otro.

Lo que hace la directiva .BYTE

Cuando se utiliza la directiva .BYTE en un programa creado con el ensamblador MAC/65 o el Atari Assembler Editor, los bytes que vienen a continuación de la directiva son ensamblados en posiciones consecutivas de la memoria RAM. En el programa llamado "The Visitor", los bytes que vienen a continuación de la etiqueta TEXT corresponden al código ATASCII (código ASCII de Atari) de una serie de caracteres de texto.

Dando vueltas en el ciclo

Como explicamos en el capítulo 6, los registros X e Y del chip 6502 pueden ser progresivamente incrementados o disminuídos dentro de los ciclos de un programa. En el programa The Visitor, el registro X es incrementado desde 0 hasta 23 dentro del ciclo en que se leen los caracteres de una cadena de texto. Los caracteres a ser leídos están escritos en código ATASCII en las líneas 100 y 120 del programa. En la línea 160, la orden LDX# 0 se utiliza para cargar un cero en el registro X. A continuación, en la línea 170, comienza el ciclo.

Incrementando el registro X

La primera instrucción dentro del ciclo es LDA TEXT, X. Cada vez que comienza el ciclo, esta instrucción usa el direccionamiento indexado para cargar el acumulador con el código ATASCII de un carácter. Luego, en la línea 180, se usa de nuevo el modo de direccionamiento indexado, esta vez para almacenar el carácter en un buffer de texto. Cuando termina el ciclo, todos los caracteres en el buffer de texto se imprimen en la pantalla. La primera vez que el programa The Visitor llegue a la línea 170, habrá un 0 en el registro X (Ya que se acaba de cargar en la línea anterior un 0 en el registro X). Así que la primera vez que el programa se encuentra con la orden LDA TEXT, X, carga el acumulador con el número hexadecimal $54 - lo que los programadores a veces llaman "el 0-avo byte" a continuación de la etiqueta TEXT. (Por cierto, no hay necesidad de poner un símbolo "#" delante del número $54, ya que los números que vienen a continuación de la directiva .BYTE siempre son interpretados como números literales por el ensamblador MAC/65 y por el Atari Assembler Editor).

Incrementando y decrementando los registros X e Y

Ahora vamos a pasar a la línea 190. El mnemónico que ve ahí - INX - significa "incrementar el registro X". Dado que el registro X contiene actualmente un 0, esta instrucción incrementará ahora ese 0 a 1. A continuación, en la línea 200, vemos la instrucción CPX #23. Esto significa "comparar el valor en el registro X con el número literal 23". La razón por la que queremos hacer esta comparación es para poder determinar si ya se han impreso 23 caracteres en la pantalla. Hay 23 caracteres en la cadena de texto que

se está imprimiendo, y cuando se hayan impreso todos, necesitaremos imprimir un retorno de carro y terminar nuestro programa.

Comparación de valores en Lenguaje Ensamblador

Hay tres instrucciones de comparación en el lenguaje ensamblador del 6502: CMP, CPX, y CPY. CMP significa "Comparar con el valor que tiene el acumulador". Cuando se usa la instrucción CMP, seguido de un operando, el valor expresado por el operando se resta al valor que tiene el acumulador. Esta operación de resta no se realiza para determinar la diferencia exacta entre estos dos valores, sino simplemente para comprobar si son o no iguales, y si no son iguales, para determinar cuál es más grande que el otro. Si el valor del acumulador es igual al valor que se desea comprobar, se activa la bandera cero (Z) del registro de estado del procesador (P). Si el valor en el acumulador no es igual al valor que se desea comprobar, la bandera Z quedará desactivada.

Si el valor en el acumulador es menor que el valor a comprobar, entonces la bandera de acarreo (C) del registro P será desactivada. Y si el valor en el acumulador es mayor o igual al valor a comprobar, entonces la bandera Z se activará y la bandera de acarreo también. CPX y CPY trabajan de la misma manera que CMP, solo que se utilizan para comparar valores con el contenido de los registros X e Y. Producen los mismos efectos que CMP sobre las banderas del registro de estado P.

Uso en conjunto de las instrucciones de comparación y de salto

Las tres instrucciones de comparación del lenguaje ensamblador de Atari se suelen utilizar en conjunto con otras ocho instrucciones - las ocho instrucciones de salto condicional que se mencionaron en el capítulo 6. El programa de ejemplo que hemos denominado The Visitor contiene una instrucción de salto condicional en la línea 210. Esa instrucción es BNE LOOP, que significa "salte a la instrucción cuya etiqueta es LOOP sólo si la bandera cero (del registro de estado del procesador) está activa". Esta instrucción usa una convención un poco confusa del chip 6502. En el registro de estado del procesador 6502, la bandera cero se activa (es igual a 1) si el resultado de la operación que se acaba de realizar es 0, y se desactiva (es igual a 0) si el resultado de la operación que se acaba de realizar es distinto de cero.

En realidad no importa

Todo esto es muy académico, sin embargo, en lo que al resultado de la orden BNE LOOP se refiere. Cuando el computador se encuentre con la línea 210, seguirá saltando de vuelta a la línea 170 (la línea con la etiqueta LOOP), siempre y cuando el valor del registro X no haya disminuído hasta llegar a cero. Una vez que el valor del registro X haya disminuído hasta llegar a cero, la instrucción BNE LOOP en la línea 210 será ignorada, y el programa pasará a la línea 220, que es la línea siguiente. En la línea 220, el programa salta a la subrutina OPNSCR - que actualmente reside en RAM a partir de la dirección de

memoria $5041, siempre y cuando el código objeto, tanto de PRNTSC como el de VISITOR, hayan sido cargados en su computador y estén listos para ser ejecutados.

Instrucciones de salto condicional

Como señalamos en el capítulo anterior, en el lenguaje ensamblador del 6502 tenemos ocho instrucciones de salto condicional. Todas comienzan con la letra B, y se les conoce también como instrucciones de direccionamiento relativo o de bifurcación. A continuación se presentan estas 8 instrucciones y su significado:

BCC – "Branch if Carry flag is Clear": Salte sólo si la bandera de acarreo (C) del registro de estado del procesador (P) está desactivada. (Si la bandera de acarreo está activada, la operación no tendrá ningún efecto).

BCS – "Branch if Carry flag is Set": Salte sólo si la bandera de acarreo (C) del registro de estado del procesador (P) está activada. (Si la bandera de acarreo está desactivada, la operación no tendrá ningún efecto).

BEQ – "Branch if EQual to zero": Salte sólo si el resultado de una operación es igual a cero (Si la bandera cero (Z) fue activada como resultado de una operación).

BMI – "Branch on MInus": Salte sólo si el resultado de una operación es menor que cero (Si la bandera de negativo (N) fue activada como resultado de una operación).

BNE – "Branch if Not Equal to zero": Salte sólo si el resultado de una operación es distinto de cero (Si la bandera de cero (Z) fue desactivada como resultado de una operación).

BPL – "Branch on PLus": Salte sólo si el resultado de una operación es mayor que cero (Si la bandera de negativo (N) fue desactivada como resultado de una operación).

BVC – "Branch if oVerflow flag is Clear": Salte sólo si la bandera de desbordamiento (V) fue desactivada como resultado de una operación.

BVS – "Branch if oVerflow flag is Set": Salte sólo si la bandera de desbordamiento (V) fue activada como resultado de una operación.

Cómo se utilizan las Instrucciones de salto condicional

El método habitual para utilizar una instrucción de salto condicional en el lenguaje ensamblador del 6502 consiste en cargar el registro X o Y con un cero o algún otro valor, y luego cargar en el registro A (o algún otro registro de la memoria) un valor a ser comparado. Una vez comparado, se usa una instrucción de salto condicional para decirle al computador cuál bandera del registro P se debe consultar, y qué hacer si dicha comprobación resulta exitosa o no. Todo esto suena muy complicado, y lo es. Pero una

vez que entienda el concepto general de salto condicional, podrá utilizar una tabla simple para escribir instrucciones de salto condicional. Aquí le presento una tabla de este tipo.

PARA COMPROBAR SI :	HAGA ESTO :	Y LUEGO ESTO OTRO :
A = VALOR	CMP #VALOR	BEQ
A <> VALOR	CMP #VALOR	BNE
A >= VALOR	CMP #VALOR	BCS
A > VALOR	CMP #VALOR	BEQ y luego BCS
A < VALOR	CMP #VALOR	BCC
A = [DIRECC]	CMP $DIRECC	BEQ
A <> [DIRECC]	CMP $DIRECC	BNE
A >= [DIRECC]	CMP $DIRECC	BCS
A > [DIRECC]	CMP $DIRECC	BEQ y luego BCS
A < [DIRECC]	CMP $DIRECC	BCC
X = VALOR	CPX #VALOR	BEQ
X <> VALOR	CPX #VALOR	BNE
X >= VALOR	CPX #VALOR	BCS
X > VALOR	CPX #VALOR	BEQ y luego BCS
X < VALOR	CPX #VALOR	BCC
X = [DIRECC]	CPX $DIRECC	BEQ
X <> [DIRECC]	CPX $DIRECC	BNE
X >= [DIRECC]	CPX $DIRECC	BCS
X > [DIRECC]	CPX $DIRECC	BEQ y luego BCS
X < [DIRECC]	CPX $DIRECC	BCC
Y = VALOR	CPY #VALOR	BEQ
Y <> VALOR	CPY #VALOR	BNE
Y >= VALOR	CPY #VALOR	BCS
Y > VALOR	CPY #VALOR	BEQ y luego BCS
Y < VALOR	CPY #VALOR	BCC
Y = [DIRECC]	CPY $DIRECC	BEQ
Y <> [DIRECC]	CPY $DIRECC	BNE
Y >= [DIRECC]	CPY $DIRECC	BCS
Y > [DIRECC]	CPY $DIRECC	BEQ y luego BCS
Y < [DIRECC]	CPY $DIRECC	BCC

Ciclos en el Lenguaje Ensamblador

En el lenguaje ensamblador del 6502, las instrucciones de comparación y de salto condicional generalmente se utilizan juntas. En el programa de ejemplo llamado The Visitor, la instrucción de comparación CPX y la instrucción de salto BNE se usan juntas en un ciclo controlado por el incremento de un valor en el registro X. Cada vez que el programa pasa por este ciclo, el valor en el registro X es incrementado o disminuido progresivamente. Y cada vez que el programa llega a la línea 200, el valor en el registro X es comparado con el número literal 23. Cuando se llega a ese número, el ciclo termina. El programa por lo tanto seguirá volviendo a la línea 170 hasta que se hayan impreso 23 caracteres en la pantalla. Luego, en las líneas 220 y 230, se abre la pantalla de su computador - limpiándola en el proceso - y se imprime la cadena que se ha transferido en

el buffer de texto. Por último, en la línea 240, el programa entrará en lo que se conoce como un bucle de ciclo infinito - saltando una y otra vez a la misma instrucción JMP, sin hacer nada más hasta que se pulsa la tecla BREAK o de alguna otra manera se detiene el programa.

¿Por qué utilizar un buffer?

Antes de pasar a nuestro siguiente tema - mejorar el programa The Visitor – vale la pena responder a una pregunta que se le puede o no haber ocurrido. La pregunta es: ¿Por qué se debe utilizar un buffer de texto? ¿Por qué no imprimir el texto en las líneas de la 100 a 120 directamente en la pantalla, sin tener que primero moverlo a un buffer y luego sacarlo del mismo?

He aquí la respuesta a esa pregunta: El texto de un buffer puede ser cargado de muchas maneras: por medio de un teclado o un módem telefónico, por ejemplo, o con datos entregados directamente por un programa. Y una vez que una cadena de texto está en el buffer, puede ser eliminado de la memoria de diferentes maneras. Otra de las ventajas de un buffer de texto es que puede ser cargado en la memoria RAM, tomar nota de su dirección, y usarlo a partir de entonces cuando lo necesite. Un buffer puede servir como un repositorio central para cadenas texto, al que luego se podrá acceder con gran facilidad y de muchas maneras.



Mejorando el Programa The Visitor

Ahora estamos listos para hacer algunas mejoras al programa llamado "The Visitor". No es que el programa no funcione, de hecho lo hace, pero tiene ciertas limitaciones. Y algunas de esas limitaciones podrían ser mejoradas fácilmente - como se ha hecho en este nuevo programa, al que he llamado Response.

Response es similar a The Visitor – pero es, como pronto veremos, significativamente mejor en varios aspectos:

RESPONSE

```
10      ;
20      ; RESPUESTA
30      ;
40      TXTBUF=$5041
50      OPNSCR=$5031
60      PRNTLN=$5031
70      ;
80      EOL=$9b
90      ;
100             *=$650
110     ;
120     TEXT    .BYTE "YO SOY vuestro líder, tonto!",EOL
130     ;
140     RSPONS
150     ;
160             LDX #0
170     LOOP
180             LDA TEXT, X
190             STA TXTBUF, X
200             CMP #$9B
210             BEQ FINI
220             INX
230             JMP LOOP
240     FINI
250             JSR OPNSCR
260             JSR PRNTLN
270     INFIN
280             JMP INFIN
```


Si desea ejecutar el programa Response - y espero que lo quiera hacer - puede digitarlo en la memoria de su computador en este momento. Puesto que llama a las mismas subrutinas que su predecesor, puede ensamblarlo y ejecutarlo tan pronto como lo haya digitado, siempre y cuando todavía tenga el programa PRNTSC cargado en la memoria RAM.

Para ejecutar el programa Response, puede llamar a su programa de depuración y usar el comando G, o ir al modo DOS y utilizar el comando de DOS apropiado. Sea cual sea el

modo que decida utilizar, será capaz de ejecutar el programa usando la dirección de ejecución $066B, siempre y cuando lo haya digitado y ensamblado de acuerdo con las sugerencias que he proporcionado. Sin embargo, incluso si usted ha seguido las instrucciones, aun así encontraremos un pequeño problema. Sólo los primeros 23 caracteres de la cadena "YO SOY vuestro líder, tonto!" se mostrarán en la pantalla del computador. Esto porque el buffer de texto que hemos creado en el programa PRNTSC tiene sólo 23 caracteres de longitud.

Este error es fácil de remediar. Pero antes de solucionarlo, tal vez sea una buena idea guardar Response en un disco, tanto en sus versiones código fuente y código objeto. Los nombres de archivo sugeridos para el programa son RESPONSE.SRC y RESPONSE.OBJ.

Arreglando el programa PRNTSC

Ahora estamos listos para alargar el buffer de texto utilizado en el programa PRNTSC, por lo que se imprimirá su mensaje completo en la pantalla del computador. Para alargar el buffer de impresión, ponga su ensamblador en modo de edición y cargue el código fuente del programa en la memoria de su computador. Luego cambie la línea 70 de BUFLEN = 23 a BUFLEN = 40. Cuando haya hecho este cambio, puede guardar el código fuente modificado bajo el nombre de archivo PRNTSC.SR2, ensamblarlo, y guardar el código objeto con el nombre de PRNTSC.OB2 (para distinguirlo de PRNTSC.SRC y PRNTSC.OBJ, los programas originales de PRNTSC).

Cuando haya guardado sus programas PRNTSC.SR2 y PRNTSC.OB2, puede volver a cargar el programa Response y ejecutarlo con PRNTSC.OB2 en lugar de PRNTSC.OBJ. Esta vez debería ver la cadena completa que el programa Response presenta en la pantalla de video, pero lamentablemente, ahora se dará cuenta de que otra cosa anda mal. A continuación de la línea "YO SOY vuestro líder, tonto!" verá una línea de pequeños corazones. ¿Cómo llegaron allí? Responderé a esa pregunta en un momento. Pero primero, echemos un vistazo a algunas de las diferencias entre el programa que se llama The Visitor y el que se llama Response.

Una rutina mejor

Desde un punto de vista técnico, el programa Response es mejor que The Visitor - por varias razones. La diferencia más obvia entre ambos programas es la forma en que manejan las cadenas de texto. En el programa llamado The Visitor, se utilizó una cadena de texto compuesta de código ATASCII. En Response, usamos un string compuesto de caracteres literales. Eso hizo que el programa sea mucho más fácil de escribir, y también mucho más fácil de leer.

Otra diferencia importante entre nuestro último programa y su predecesor es la forma en que se escribió el bucle. En el programa llamado The Visitor, el ciclo cuenta el número de

caracteres que se han impreso en la pantalla, y termina cuando el conteo llega a 23. Ahora bien, este es un sistema perfectamente correcto - para imprimir cadenas de texto que tienen 23 caracteres de longitud. Desafortunadamente, no es tan apropiado para imprimir cadenas de otras longitudes. Por lo que no es una rutina muy versátil para imprimir caracteres en la pantalla.

Comprobando un Retorno de Carro

El programa Response es mucho más versátil que The Visitor, ya que puede imprimir cadenas de casi cualquier longitud en una pantalla. Esto porque no hace un seguimiento del número de caracteres que se han impreso en la pantalla. En vez de eso, cada vez que el programa encuentra un carácter, comprueba para ver si su valor es $9B - el código ATASCII del retorno de carro, o el carácter de final de línea (EOL). Si el carácter es no es un EOL, el computador lo imprime en la pantalla y pasa al siguiente carácter en la cadena. Si el carácter es un EOL, se imprime en la pantalla y la rutina termina. Y eso es todo - excepto por esos corazones un poco molestos que nos encontramos cuando ejecutamos el programa Response. Ahora vamos a echar otro vistazo a los corazones, y veremos qué podemos hacer al respecto.



Una Cadena de Corazones

Los corazones están allí porque el buffer de texto que hemos creado para nuestro mensaje de los programas Visitor y Response tiene ahora 40 caracteres de largo - más largo que cualquiera de nuestros mensajes. Y la parte sobrante del buffer está rellena con ceros, tal como generalmente están las posiciones de memoria vacías de un computador. ¿Entonces por qué los corazones? Bueno, en el código de caracteres ATASCII que utiliza su Atari, un cero no equivale a un espacio; en cambio, equivale a un carácter gráfico con forma de corazón. El código ASCII para el espacio es $20, o 32 en la notación decimal. Basta con mirar la cadena de caracteres ASCII en los programas Visitor y Response, y verá que los espacios en el mensaje "LLEVAME CON TU LIDER!" son de hecho representados por el valor $20.

Es posible, por supuesto, imprimir los mensajes en la pantalla de su Atari sin cadenas de corazones a continuación. Lo que tiene que hacer para evitar que aparezcan estos

corazones es limpiar su buffer de texto - o, más exactamente, rellenarlo con espacios - antes de que se ejecute el programa.

Limpieza del Buffer de Texto

He aquí una rutina corta que limpiará un buffer de texto - o cualquier otro bloque de memoria - y lo rellenará con espacios, ceros, o cualquier otro valor que usted elija. Mediante la incorporación de esta rutina en nuestros programas The Visitor y Response, puede reemplazar los ceros de sus mensaje que aparecerá en pantalla por espacios ASCII, y hacer que los espacios aparezcan como tales en la pantalla de su computador, en vez de corazones. A medida que siga trabajando con el lenguaje ensamblador, encontrará que las rutinas de borrado de memoria como ésta pueden ser muy útiles en muchos tipos de programas. Procesadores de texto, programas de telecomunicaciones, y muchos tipos de paquetes de software hacen uso extensivo de rutinas que limpian los valores en los bloques de memoria y los reemplazan por otros valores.

PROGRAMA PARA LIMPIAR UN BLOQUE DE MEMORIA

```
1300  FILL
1310        LDA #FILLCH
1320        LDA #BUFLEN
1330  START
1340        DEX
1350        STA TXTBUF,X
1360        BNE START
1370        RTS
```

Este programa es muy sencillo. Usa el direccionamiento indirecto y una cuenta atrás sobre el registro X, para llenar cada dirección de memoria en un buffer de texto (TXTBUF) con un carácter de relleno (FILLCH) determinado. Luego, el programa termina. Esta rutina funciona con cualquier carácter de relleno de 8 bits, y con cualquier longitud de buffer (BUFLEN) de hasta 255 caracteres. Más adelante en este libro encontrará algunas rutinas de 16 bits que pueden rellenar con valores bloques más extensos de memoria RAM. Puede utilizar esta rutina, integrándola en sus programas The Visitor y Response. Agreguémosla a estos dos programas ahora, comenzando con The Visitor. Con su ensamblador en modo de edición, cárguelo desde el disco y agréguele las siguientes líneas:

```
55    BUFLEN=40
60    FILLCH=$20
150         JSR FILL
1300  FILL
1310        LDA #FILLCH
1320        LDA #BUFLEN
1330  START
1340        DEX
1350        STA TXTBUF,X
1360        BNE START
```

```
1370        RTS
```

Cuando estos cambios se hayan hecho, su programa VISITOR.SRC debería tener el siguiente aspecto:

THE VISITOR

```
10    ;
20    ; EL VISITANTE
30    ;
35    TXTBUF=$5041
40    OPNSCR=$5003
50    PRNTLN=$5031
55    BUFLEN=40
60    FILLCH=$20
70    ;
80            *=$600
90    ;
0100  TEXT    .BYTE $4C,$4C,$45,$56,$41,$4D,$45,$20
0110          .BYTE $43,$4F,$4E,$50,$20,$54,$55,$20
0120          .BYTE $4C,$49,$44,$45,$52,$21
0130  ;
0140  VIZTOR
0150          JSR FILL
0160          LDX #0
0170  LOOP    LDA TEXT,X
0180          STA TXTBUF,X
0190          INX
0200          CPX #23
0210          BNE LOOP
0220          JSR OPNSCR
0230          JSR PRNTLN
0240  INFIN   JMP INFIN
1300  FILL
1310          LDA #FILLCH
1320          LDX #BUFLEN
1330  START
1340          DEX
1350          STA TXTBUF,X
1360          BNE START
1370          RTS
```

Cuando su programa se vea así, puede guardar en un disco su versión mejorada. Entonces puede hacer exactamente los mismos cambios en su programa Response, y guardarlo también. Cuando haya terminado con el programa Response, debería verse así:

RESPONSE

```
10    ;
20    ; RESPUESTA
30    ;
40    TXTBUF=$5041
```

```
50      OPNSCR=$5031
60      PRNTLN=$5031
65      BUFLEN=40
70      FILLCH=$20
75      ;
80      EOL=$9b
90      ;
100             *=$650
110     ;
120     TEXT    .BYTE "YO SOY vuestro líder, tonto!",EOL
130     ;
140     RSPONS
150     ;
160             LDX #0
170     LOOP
180             LDA TEXT, X
190             STA TXTBUF, X
200             CMP #$9B
210             BEQ FINI
220             INX
230             JMP LOOP
240     FINI
250             JSR OPNSCR
260             JSR PRNTLN
270     INFIN
280             JMP INFIN
1300    FILL
1310            LDA #FILLCH
1320            LDX #BUFLEN
1330    START
1340            DEX
1350            STA TXTBUF,X
1360            BNE START
1370            RTS
```

Haciéndolo

Cuando haya guardado de manera segura en un disco ambos programas mejorados - tanto en sus versiones código fuente como código objeto - podrá ejecutarlos y verá que todos nuestros problemas con nuestro buffer de texto se habrán resuelto. Y eso nos lleva a nuestro próximo capítulo, en el que aprenderá a llamar a rutinas de lenguaje ensamblador desde programas en BASIC.

Capítulo Ocho
Invocando Programas en Lenguaje Ensamblador desde el BASIC

A veces es difícil decidir si un programa debe ser escrito en BASIC o en lenguaje ensamblador. Pero no siempre es necesario tomar esta decisión. En muchos casos, puede combinar la simplicidad del BASIC con la velocidad y la versatilidad del lenguaje ensamblador simplemente escribiendo rutinas de lenguaje ensamblador que se pueden llamar desde el BASIC, eso es lo que vamos a aprender en este capítulo.

Los dos primeros programas con los que vamos a trabajar son los que se introdujeron en el Capítulo 7: los programas llamados The Visitor y Response. Sin embargo, antes de que podamos llamar a estos dos programas desde el BASIC, tendremos que hacer un par de cambios en la forma en que fueron escritos. En primer lugar, vamos a tener que eliminar el bucle infinito al final de cada programa, y sustituirlo por la instrucción RTS, de modo que cada programa vuelva a BASIC cuando haya terminado, en vez de quedar en un bucle infinito. También tendrá que añadir una instrucción PLA en el comienzo de cada programa para que no estropeen la pila cuando se les llama desde el BASIC. Más adelante en este capítulo vamos a explicar exactamente por qué son necesarias estas instrucciones PLA. Si acaba de terminar el capítulo 7 y todavía tienen su computador encendido, entonces puede cargar los programas The Visitor y Response y hacer los cambios necesarios en ellos.

Comenzando con The Visitor

Vamos a empezar con el programa llamado The Visitor. Con su ensamblador funcionando y en modo EDIT, cargue el código fuente de The Visitor en el computador y teclee LIST. Esto es lo que debería ver:

THE VISITOR

```
10    ;
20    ;THE VISITOR
30    ;
35    TXTBUF=$5041
40    OPNSCR=$5003
50    PRNTLN=$5031
55    BUFLEN=40
60    FILLCH=$20
70    ;
80            *=$0600
90    ;
0100  TEXT    .BYTE $4C,$4C,$45,$56,$41,$4D,$45,$20
0110          .BYTE $43,$4F,$4E,$20,$54,$55,$20
```

```
0120          .BYTE $4C,$49,$44,$45,$52,$21
0130  ;
0140  VIZTOR
0150          JSR FILL
0160          LDX #0
0170  LOOP    LDA TEXT,X
0180          STA TXTBUF,X
0190          INX
0200          CPX #23
0210          BNE LOOP
0220          JSR OPNSCR
0230          JSR PRNTLN
0240  INFIN   JMP INFIN
1300  FILL
1310          LDA #FILLCH
1320          LDX #BUFLEN
1330  START
1340          DEX
1350          STA TXTBUF,X
1360          BNE START
1370  RTS
```

Ahora puede modificar el programa para que pueda ser llamado desde el BASIC. En primer lugar, inserte una línea que contenga una instrucción PLA digitando

```
145  PLA
```

A continuación, cambie el ciclo infinito en la línea 240 por una instrucción RTS escribiendo

```
240  RTS
```

Cuando haya terminado, le sugiero que cambie el nombre del programa por VISITOR.SR2 para diferenciarlo del programa original. Para ello, escriba

```
20 ;VISITOR.SR2
```

y presione la tecla RETURN.
Listo.

Cuando haya hecho todos los cambios en el programa VISITOR.SRC, escriba LIST de nuevo y esto es lo que debería ver:

THE VISITOR

```
10    ;
20    ;VISITOR.SRC2
30    ;
35    TXTBUF=$5041
40    OPNSCR=$5003
```

```
50    PRNTLN=$5031
55    BUFLEN=40
60    FILLCH=$20
70    ;
80            *=$0600
90    ;
0100  TEXT    .BYTE $4C,$4C,$45,$56,$41,$4D,$45,$20
0110          .BYTE $43,$4F,$4E,$20,$54,$55,$20
0120          .BYTE $4C,$49,$44,$45,$52,$21
0130  ;
0140  VIZTOR
0145          PLA
0150          JSR FILL
0160          LDX #0
0170  LOOP    LDA TEXT,X
0180          STA TXTBUF,X
0190          INX
0200          CPX #23
0210          BNE LOOP
0220          JSR OPNSCR
0230          JSR PRNTLN
0240          RTS
1300  FILL
1310          LDA #FILLCH
1320          LDX #BUFLEN
1330  START
1340          DEX
1350          STA TXTBUF,X
1360          BNE START
1370          RTS
```

Guardando su programa corregido

Cuando haya hecho todos los cambios necesarios en el código de su programa The Visitor, guárdelo nuevamente en un disco - tanto en sus versiones fuente como objeto - usando los nombres de archivo VISITOR.SR2 y VISITOR.OB2. Entonces estará listo para modificar el otro programa que le presentamos en el Capítulo 7 - Response - de modo que también pueda ser llamado desde el BASIC.

Modificando el programa Response

Para arreglar el programa RESPONSE.SRC de modo que se pueda acceder desde el BASIC, sólo tiene que cargar el código fuente en el computador y hacer que los mismos tres cambios que hizo en el programa The Visitor. Cuando los haya realizado, Response debería verse así:

RESPONSE.SRC

```
10    ;
20    ; RESPONSE
30    ;
40    TXTBUF=$5041
50    OPNSCR=$5003
60    PRNTLN=$5031
65    BUFLEN=40
70    FILLCH=$20
75    ;
80    EOL=$9B
90    ;
0100         *=$0650
0110 ;
0120 TEXT    .BYTE " YO SOY vuestro líder, tonto!",EOL
0130 ;
0140 RSPONS
0145         PLA
0150         JSR FILL
0160         LDX #0
0170 LOOP
0180         LDA TEXT,X
0190         STA TXTBUF,X
0200         CMP #$9B
0210         BEQ FINI
0220         INX
0230         JMP LOOP
0240 FINI
0250         JSR OPNSCR
0260         JSR PRNTLN
0270         RTS
1300 FILL
1310         LDA #FILLCH
1320         LDX #BUFLEN
1330 START
1340         DEX
1350         STA TXTBUF,X
1360         BNE START
1370         RTS
```

Ejecutándolo

Cuando haya realizado los cambios necesarios en el programa Response, puede guardar sus nuevas versiones bajo los nombres de archivo RESPONSE.SR2 y RESPONSE.OB2. Entonces estaremos listos para llamar a los programas The Visitor y Response desde el BASIC. Para ello, necesitará un cartucho de BASIC en su computador, si éste es un Atari 400, 800 o un 1200XL. (No necesita un cartucho de BASIC si tiene uno de los modelos actualizados de computador Atari, ya que éstos traen BASIC incorporado.) Cuando tenga su computador funcionando nuevamente, ejecutando el BASIC, y con el disco de datos en la unidad de disco, lo primero que tendrá que hacer es llamar al menú del DOS de Atari.

Cuando éste haya cargado, seleccione la opción L del menú (carga binaria) y cargue en la memoria de su computador los programas VISITOR.OB2, RESPONSE.OB2 y PRNTSC.OB2.

Cuando haya cargado los tres programas, seleccione la opción B para devolver el control al intérprete BASIC de computador. Luego, cuando en su pantalla aparezca el mensaje READY del intérprete BASIC, escriba el siguiente comando BASIC:

```
X=USR(1559)
```

Si ha escrito y ensamblado su programa VISITOR.SR2 de la misma manera en que lo hicimos anteriormente, y si su código objeto está almacenado en la memoria de su computadora, entonces éste debería ejecutarse tan pronto como escriba la orden X = USR (1559) y presione la tecla RETURN, ya que su dirección de comienzo es $0617, o 1559 en decimal.

Del mismo modo, puede ejecutar RESPONSE.OB2 simplemente escribiendo

```
X=USR(1643)
```

Y presionando la tecla RETURN.

Ahora que ya ha invocado dos programas desde el BASIC, estamos listos para ver cómo lo hizo: en concreto, cómo él trabaja la función USR del BASIC de Atari, y cómo se utiliza en la programación en lenguaje ensamblador.

La función USR

Los programas de lenguaje de máquina, como acabamos de señalar, son llamados desde el BASIC por medio de una función especial llamada USR. La función USR puede ser escrita de dos maneras: sin argumentos, o con uno o más argumentos opcionales. Una llamada que no incluye argumentos se escribe con el formato que utilizamos para llamar a nuestros programas The Visitor y Response:

```
X=USR(1643)
```

Cuando se escribe una llamada utilizando este formato, el número entre paréntesis equivale a la dirección inicial (expresada como un número decimal) del programa de lenguaje de máquina que va a ser llamado. Cuando el programa en lenguaje de máquina termine su ejecución, el control del equipo será devuelto al BASIC. Si un programa está en ejecución y éste utiliza la función USR, el programa se reanudará en la primera instrucción que viene a continuación de la función USR, una vez que el control sea devuelto al BASIC.

Funciona como "GOSUB"

En este sentido, la orden USR funciona como una instrucción GOSUB común. Como una subrutina escrita en BASIC, un programa en lenguaje de máquina llamado desde el BASIC debe terminar con una instrucción de retorno. En BASIC, la instrucción de retorno es, lógicamente, RETURN. En el lenguaje ensamblador, la instrucción de retorno es RTS, que significa "ReTurn from Subroutine - Retorno de la subrutina". Una llamada en la que se utilizan argumentos se ve más o menos así:

```
X=USR(1536,ADR(A$),ADR(B$))
```

O así:

```
X=USR(1536,X,Y)
```

Cuando se usan argumentos en la función USR, cada argumento puede ser cualquier valor que equivalga a un número de 16 bits. Se pueden realizar operaciones de lenguaje de máquina sobre los valores referidos en los argumentos, y los resultados de esas operaciones se pueden pasar de vuelta al BASIC cuando se terminen las operaciones en lenguaje de máquina. De esta manera, se pueden usar en programas BASIC operaciones que requieren ejecutarse en velocidad lenguaje de máquina. Ya sea si se usan o no argumentos en una función USR, un programa en lenguaje de máquina llamado por la función USR también puede devolver un valor de 16 bits al BASIC, cuando le pase el control a éste.

Volviendo al BASIC

Para devolver un valor al BASIC cuando un programa en lenguaje de máquina haya terminado, todo lo que tiene que hacer es almacenar el valor que va a ser devuelto en dos posiciones de memoria especiales de 8-bits, antes de retornar el control al BASIC. Estas dos posiciones son $D4 y $D5 (212 y 213 en notación decimal). Cuando un valor que va a ser devuelto al BASIC se almacena en estas dos posiciones, debe ser almacenado con el byte menor en $D4 y el byte mayor en $D5. Cuando se devuelva el control al BASIC, el valor de 16 bits en las dos posiciones será convertido automáticamente a un valor decimal entre 0 y 65535. Luego ese valor será devuelto al BASIC como el valor de la variable en la orden USR, por ejemplo, la "X" en la orden X = USR (1536, X, Y).

Cómo trabaja la función USR

Cuando un programa BASIC encuentra una orden USR, se coloca en la parte superior de la pila la dirección de memoria de la instrucción actual del programa en ejecución. Luego, se pone en la pila un número de 8 bits correspondiente al número de argumentos que aparecen en la orden USR. Si no hay argumentos en la orden USR, se coloca un cero en la parte superior de la pila. Cuando una orden USR llama a una subrutina en lenguaje de máquina, la dirección de retorno de la subrutina en lenguaje de máquina siempre es

"tapada" en la pila por otro número y, por lo tanto, éste debe ser eliminado de la pila (incluso si es un cero) antes de que el control sea devuelto al BASIC.

Limpiando la Pila

Y es por eso que el primer mnemotécnico en la mayoría de los programas de lenguaje de máquina diseñados para ser llamados desde el BASIC es una instrucción para "limpiar la pila": la instrucción PLA.

Más operaciones de la Pila

Si se incluyen los argumentos en la función USR, entonces se requieren más operaciones sobre la pila, cuando se tenga que llamar a un programa de lenguaje de máquina. Antes de que el número de argumentos se coloque en la pila, los argumentos mismos son insertados en la pila. Luego, cuando el programa en lenguaje de máquina comienza su ejecución, los argumentos pueden ser eliminados de la pila y ser procesados por el programa en lenguaje de máquina de cualquier forma que se desee. Por último, cuando termina el programa en lenguaje de máquina y el control del computador pasa de nuevo al BASIC, los resultados de las operaciones en lenguaje de máquina que se hayan realizado con los argumentos de la orden USR pueden ser almacenados en las posiciones memoria $D4 y $D5. Los valores que se hayan almacenado en este par de posiciones pueden ser devueltos al BASIC como el valor de la variable usada en la orden USR original.

Un programa de ejemplo

Esto puede sonar un poco complicado, pero un programa de ejemplo que pueda escribir en su computador y ejecutarlo desde el BASIC aclarará las cosas. Escriba el siguiente programa en su computador.

UN PROGRAMA DE SUMAS DE 16 bits

```
10    ;
20    NUM1 = $CB
30    NUM2 = $CE
40    SUM = $D4
50    ;
60         *=$0600
70    ;
90         CLD
0100       PLA ;LIMPIAR # DE ARGUMENTOS DEL ACUMULADOR
0105       PLA
0110       STA NUM1+1 ;BYTE MAYOR DEL PRIMER ARGUMENTO
0120       PLA
0130       STA NUM1 ;BYTE MENOR DEL PRIMER ARGUMENTO
0140  PLA
0150       STA NUM2+1 ;BYTE MAYOR DEL SEGUNDO ARGUMENTO
0160       PLA
```

```
0170       STA NUM2 ;BYTE MENOR DEL SEGUNDO ARGUMENTO
0180  ;
0190       CLC
0210       LDA NUM1 ;BYTE MENOR DE NUM1
0220       ADC NUM2 ;BYTE MENOR DE NUM2
0230       STA SUM  ;BYTE MENOR DE SUM
0240       LDA NUM1+1 ;BYTE MAYOR DE NUM1
0250       ADC NUM2+1 ;BYTE MAYOR DE NUM2
0260       STA SUM+1 ;BYTE MAYOR DE SUM
0270       RTS
```

Cuando haya terminado de digitar el programa, ensámblelo y guarde el código en sus versiones fuente y objeto. Los nombres de archivo sugeridos son "ADD16B.SRC" y "ADD16B.OBJ".

Una Máquina de Sumas

Cuando haya guardado sus programas, saque el ensamblador de su computador, y digite el siguiente programa BASIC:

LA MAQUINA DE SUMAR MAS CARA DEL MUNDO

```
10  GRAPHICS 0:PRINT:PRINT "LA MAQUINA DE SUMAR MAS CARA DEL
    MUNDO"
20  PRINT:PRINT "X=";
30  INPUT X
40  PRINT "Y=";
50  INPUT Y
60  SUM=USR(1536,X,Y)
70  PRINT "X+Y=";SUM
80  PRINT
90  GOTO 20
```

Cuando haya terminado de digitar el programa, guárdelo usando el nombre de archivo "ADD16B.BAS". Esto completa nuestros preparativos para llamar desde el BASIC a nuestro nuevo programa de sumas. Ahora puede llamar desde el BASIC a su programa ADD16B.OBJ de la misma manera que llamó a su programa Response. Sólo tiene que llamar a su menú de DOS y cargar el archivo binario ADD16B.OBJ. Presione la tecla "B" para volver al modo de edición del BASIC, cargue su archivo ADD16B.BAS, y digite "RUN". A continuación, puede comenzar a usar "LA MAQUINA DE SUMAR MAS CARA DEL MUNDO".

Al ingresar algunos números en su nueva máquina de sumar, se dará cuenta de que se pueden realizar sumas de 16 bits. Aunque no hemos cubierto todavía en el libro las sumas de 16-bits, probablemente entenderá cómo funciona el programa sin problemas.

En primer lugar, en las líneas de la 20 a 40, el programa reserva espacio de memoria para tres números de 16 bits: los dos números que se sumarán, y la suma de ambos. Los dos números que se sumarán están etiquetados como num1 y num2, y la dirección donde se almacenará la suma tiene la etiqueta SUM. En la línea 100, el programa borra la pila hardware con una instrucción PLA. Esta orden elimina el valor de 8 bits que representa el "número de argumentos" que nuestro programa BASIC ha colocado en la parte superior de la pila - los números de 16 bits que se han incluido en la orden USR de nuestro programa BASIC. Estos son los dos números que se sumarán, y nuestro programa de lenguaje de máquina ahora los coloca en los dos pares de posiciones de memoria de 8 bits cuyas etiquetas son NUM1 y NUM2.

El corazón del programa

Ahora hemos llegado a la parte principal de nuestro programa en lenguaje ensamblador. En la línea 200, el programa desactiva la bandera de acarreo, tal como debería hacerlo todo buen programa de sumas. Entonces comienza la suma propiamente tal.

El programa suma los bytes menores de NUM1 y NUM2, y almacena el resultado de este cálculo en el byte menor de SUM. A continuación, suma los bytes mayores de NUM1 y NUM2 (junto con un acarreo, si es que lo hay) y almacena el resultado de este cálculo en el byte mayor de SUM.

Eso es todo lo que hay que hacer. Como resultado, y esto no es un accidente, la dirección de memoria que se ha reservado para el valor SUM son $D4 y $D5, las dos posiciones especiales de memoria que siempre se devuelven al BASIC como el valor de la variable en una orden USR. Es un poco complicado, pero en realidad no es difícil de entender. Significa que nuestro programa en lenguaje de máquina ha resuelto la ecuación que se le presentó cuando se le entregó a la instrucción USR en nuestro programa BASIC. Esa orden USR, como usted recordará, era así:

```
SUM=USR(1536,X,Y)
```

Ahora los valores se han asignado a todas sus variables.

A la "X" y a la "Y" en las ecuaciones se les asignaron valores cuando fueron digitados durante la ejecución de la parte BASIC del programa. Luego, cuando el control de su equipo fue trasladado al lenguaje de máquina, los valores de X e Y fueron convertidos a números de 8 bits y colocados en la pila hardware. En el programa de lenguaje de máquina que usted escribió, los valores de X e Y se sacan de la pila. Luego se almacena en dos pares de posiciones de memoria de 8 bits (que tienen la etiqueta NUM1 y NUM2, para evitar cualquier confusión con los registros X e Y del 6502). Luego, los valores de X y Y (ahora llamados NUM1 y NUM2) han sido sumados. El resultado es almacenado en las direcciones $D4 y $D5. Cuando el control vuelve al BASIC, el intérprete convierte el valor en $D4 y $D5 en un valor de 16-bits y asigna ese valor a la variable SUM del BASIC, la

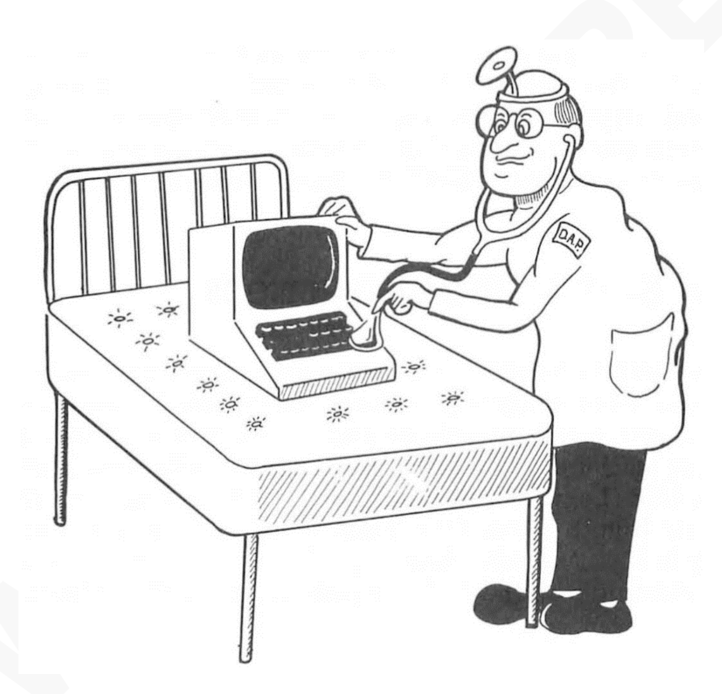
variable utilizada en la orden USR que invocó al programa en lenguaje de máquina. Entonces, tal como lo ordena el programa BASIC que escribió, el intérprete imprime el valor almacenado en la variable SUM del BASIC en la pantalla.

Esta fue una serie compleja de operaciones, como los son la mayoría de las secuencias de operaciones en lenguaje de máquina. Lamentablemente, todavía no hemos escrito un programa de sumas muy útil. Si bien es cierto que la máquina de sumar que acabamos de crear es muy cara, no es muy útil en las aplicaciones del mundo real. Puede sumar números de 16 bits e imprimir los resultados de 16-bits. Esa es una clara mejora con respecto al programa de 8 bits que creamos unos capítulos atrás, pero todavía tiene graves deficiencias. No puede manejar números o resultados que sean más largos que 16 bits. No puede trabajar con números decimales de punto flotante, o con números con signo. Si escribe un número que es demasiado grande para que lo pueda manejar, no se lo hará saber, simplemente "rodará" después del cero y sumará números sin ningún tipo de acarreo, y entregará resultados incorrectos.

Obviamente, todavía no hemos conseguido escribir un programa que trabaje como un buen programa de sumas. Ni siquiera hemos mirado programas de restas, multiplicación o división. Sin embargo, muy pronto los veremos. También vamos a discutir muchos otros temas, como números con signo, números BCD, manipulación de bits y mucho más en el capítulo 10, Matemáticas del Lenguaje Ensamblador. Antes de entrar en el capítulo 10, sin embargo, veremos que queda un poco de terreno por recorrer en Programando Bit a Bit, el tema del capítulo 9.

Capítulo Nueve
Programando Bit a Bit

En el mundo de la programación de computadores, realizar operaciones binarias en un solo bit es algo parecido a hacer microcirugía. Si es capaz de comprobar bits, desplazar y, en general manipular bits con habilidad, entonces usted es un verdadero D.P.E. (Doctor de programación en lenguaje ensamblador). Sin embargo, la manipulación de bits, como la mayoría de las facetas de la programación en lenguaje ensamblador, no es tan difícil como parece a primera vista. La comprensión de algunos principios básicos eliminará gran parte del misterio del desplazamiento, comprobación y otras operaciones de un solo bit en lenguaje ensamblador. Ya hemos visto muchos de los conceptos que necesita saber para convertirse en un cirujano experto en bits.



Por ejemplo, considere lo que ya ha aprendido sobre el uso del bit de acarreo del registro de estado del procesador del 6502. El uso el bit de acarreo es una de las técnicas de manipulación de bits más importantes en el lenguaje ensamblador del 6502. Ya tiene un poco de experiencia en el uso del bit de acarreo, esto lo vio en los programas de sumas. En este capítulo, tendrá la oportunidad de enseñarle a su computador cómo realizar algunos trucos nuevos utilizando el bit de acarreo del registro de estado (P) del procesador 6502.

Usando el bit de acarreo en Operaciones de desplazamiento de bits ("Bit-Shifting")

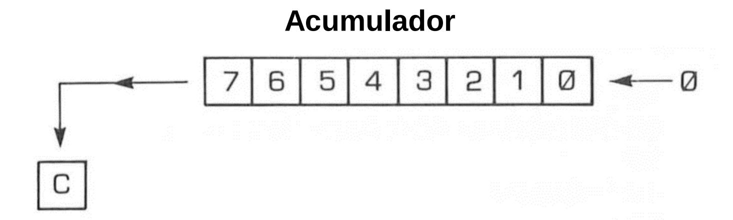
Como hemos señalado en varias ocasiones, el microprocesador 6502 de su computador Atari es un chip de 8 bits: no puede realizar operaciones con números mayores a 255 sin tener que hacer algunas contorsiones un poco tortuosas. Con el fin de procesar números mayores a 255, el 6502 debe dividirlos en trozos de 8-bits, y luego realizar las operaciones requeridas en cada trozo del número. Después, cada trozo del número debe volver a juntarse. Una vez que esté familiarizado y sepa cómo se hace esto, verá que no es tan difícil como parece. De hecho la tijera electrónica que se utiliza en todas estas operaciones electrónicas de cortar y pegar en realidad es un solo bit, el bit de acarreo en el registro de estado (P) del procesador 6502.

Cuatro instrucciones de desplazamiento de bits

Ya ha visto cómo funcionan las operaciones de acarreo en varios programas de este libro. Pero para aclarar cómo funciona el acarreo en la aritmética del 6502, será útil examinar en detalle cuatro instrucciones muy especializadas del lenguaje de máquina: ASL ("Arithmetic Shift Left - desplazamiento aritmético a la izquierda"), LSR ("Logical Shift Right - Desplazamiento lógico a la derecha"), ROL ("ROtate Left - Rotar a la izquierda"), y ROR ("ROtate Right - Rotar a la derecha"). Estas cuatro instrucciones son usadas ampliamente en el lenguaje ensamblador del 6502. Nos ocuparemos de cada uno de ellos, comenzando con la instrucción ASL (Desplazamiento aritmético a la izquierda).

ASL (Desplazamiento aritmético a la izquierda)

Como señalamos anteriormente en el capítulo sobre la aritmética binaria, cada número redondo (terminado en uno o más ceros) en notación binaria es igual al cuadrado de número binario redondo anterior. En otras palabras, 1000 000 ($80) es el doble del número 0100 000 ($40), que es el doble del número 0010 0000 ($20), que es el doble del número 0001 0000 ($ 10), y así sucesivamente. Por lo tanto, es muy fácil de multiplicar un número binario por 2. Todo lo que tiene que hacer es desplazar cada bit del número una posición a la izquierda, y colocar un cero en el bit que ha quedado vacío a causa de este desplazamiento, que es el bit 0, o el bit de más a la derecha del número. Si el bit más a la izquierda (bit 7) del número que se quiere multiplicar por 2 es un 1, entonces es necesario un acarreo. Toda esta operación que acabamos de describir, desplazar un byte a la izquierda y con acarreo, se puede realizar con una sola instrucción en 6502 en lenguaje ensamblador. Esta instrucción es ASL, que significa "Arithmetic Shift Left - Desplazamiento aritmético a la izquierda". He aquí un ejemplo de cómo funciona la instrucción ASL:



Como puede ver en la ilustración, la instrucción ASL mueve cada bit de un número de 8 bits un espacio a la izquierda, excepto el Bit 7. El Bit 7 'cae' en el bit de acarreo del registro de estados del procesador (P). La instrucción ASL se utiliza para muchos propósitos en el lenguaje ensamblador del 6502. Puede utilizarlo como una manera fácil de multiplicar un número por dos.

```
10  ;
20    *=$0600
30  ;
40    LDA #$40 ;REM 0100 0000
50    ASL A    ;DESPLAZAR A LA IZQUIERDA EL VALOR EN EL ACUMULADOR
60    STA $CB
70    .END
```

Si ejecuta este programa, y luego usa el comando "D" (“display – desplegar”) del depurador para examinar el contenido de la posición de memoria $CB, verá que el número $40 90100 0000 ha sido multiplicado por dos (quedando en $80 (1000 0000)) antes de ser almacenado en la dirección de memoria $CB.

Compresión de datos mediante el uso de ASL

Otro uso que tiene la instrucción ASL es la "compresión" de datos, y por lo tanto el aumento de la capacidad de memoria efectiva de un computador. Para tener una idea de cómo funciona la compresión de datos, supongamos que tiene una serie de valores de 4 bytes almacenados en un bloque de memoria en su computador. Estos valores pueden ser caracteres ASCII, números BCD (veremos más acerca de estos números más adelante) o cualquier otro tipo de valores de 4-bits. Utilizando la instrucción ASL, puede comprimir dos valores en un solo byte del bloque de memoria en el que están almacenados. Por lo tanto puede almacenar estos valores en la mitad del espacio de memoria que habrían ocupado en su forma normal sin comprimir. A continuación se presenta una rutina que puede utilizar en un bucle para comprimir cada byte de datos:

```
10   ;
20   ; PROGRAMA PARA COMPRIMIR DATOS
30   ;
40      *=$0600
50   ;
60   NYB1=$C0
```

```
70   NYB2=$C1
80   PKDBYT=$C2
90   ;
100    LDA #$04
110    STA NYB1
120    LDA #$06
130    STA NYB2
140  ;
150    CLC
160    LDA NYB1
170    ASL A
180    ASL A
190    ASL A
200    ASL A
210    ADC NYB2
220    STA PKDBYT
230    .END
```

Cómo funciona la rutina

Esta rutina carga un valor de 4 bits en el acumulador, desplaza ese valor al nibble (sección de 4 bits de un byte) más alto del acumulador, y luego usa la instrucción ADC para colocar otro valor de 4 bits en el nibble más bajo del acumulador. El acumulador queda entonces "empaquetado" con dos valores de 4-bits, y esos dos valores se almacenan en un único registro de memoria de 8 bits.

Comprobando los resultados

Digite el programa en su computadora, y luego use el comando G (“GOTO – ir a”) del MAC/65 o del depurador Atari para ejecutarlo. Entonces, si se ejecuta correctamente, puede usar el comando "D" (display) de su depurador para ver exactamente qué se ha hecho. Con su ensamblador en modo de depuración, digite "DC0" y verá la siguiente línea:

```
00C0 04 06 46 00 00 00 00 00
```

Como se puede ver en esta línea, el programa ha almacenado el número $04 en la dirección de memoria $C0, y $06 en dirección de memoria $C1. Ambos valores han sido comprimidos en la dirección de memoria $C1. No hace falta mucha imaginación para ver cómo esta técnica puede duplicar la capacidad de su computador para almacenar números de 4 bits en posiciones de memoria de 8 bits.

Descompresión de datos

No sería nada bueno comprimir datos si más adelante éstos no pueden ser descomprimidos. Sucede que los datos que han sido comprimidos utilizando ASL pueden

ser descomprimidos usando la instrucción complementaria LSR ("Logical Shift Right - Desplazamiento lógico a la derecha"). Hablaremos de la instrucción LSR más adelante en este capítulo.

Carga de un registro de color mediante ASL

En el lenguaje ensamblador del Atari, el comando ASL también se puede utilizar para controlar los colores de la pantalla. He aquí cómo puede hacerlo. En un computador Atari los colores que se usan en los gráficos por pantalla se almacenan en cinco registros de color. Las tablas que detallan la lista de los colores y valores de luminancia que pueden ser almacenados en estos registros se encuentran en el capítulo 9 del Manual de referencia del BASIC de Atari. El nibble superior de cada registro de color tiene un valor de color, que es el mismo número que se especifica en el segundo parámetro del comando SETCOLOR del BASIC de Atari. Los bits 1, 2 y 3 de cada registro de color ocupan el valor de luminancia del color, que es el mismo número que se especifica en el tercer parámetro del comando SETCOLOR del BASIC. No importa qué valor tiene el bit 0 de un registro de color, ya que éste no se utiliza. Mediante el uso de la instrucción ASL, se pueden controlar fácilmente los colores que aparecen en pantalla en un programa en lenguaje ensamblador de Atari.

Cómo se hace

El registro de color 2 almacena el color del fondo en Graphics 0, el modo de texto estándar del computador Atari. Suponga que quiere cargar este registro con su color estándar, el cual es azul claro. En su Atari, la dirección de memoria del registro de color 2 es la $02C6. El número de código Atari para el azul es 9, y el número de código para la luminancia del azul claro utilizado en la pantalla GRAPHICS 0 es 4. Por lo tanto se puede usar el comando ASL para almacenar el color azul claro en el Registro de Color 2 de la siguiente manera:

```
10   ;
20   ; PROGRAMA SETCLR
30   ;
40      *=$0600
50   ;
60      CLC
70      CLD
80      LDA #$09 ;REM AZUL CLARO
90      ASL A
100     ASL A
110     ASL A
120     ASL A
130     STA $02C6 ;REM REGISTRO DE COLOR 2
140     LDA #$04  ;LUMINANCIA No.4
150     ASL A
160     ADC $02C6
```

```
170    STA $02C6
180    .END
```

Como puede ver, este programa carga el Registro de Color 2 (dirección $ 02C6) con el color #$09, y luminancia #$04, o sea, el color azul claro que Atari utiliza para el fondo de su pantalla estándar de GRAPHICS 0. Si ensamblamos el programa y lo ejecutamos, estos son los valores que quedarán en cada bit del Registro de Color 2 (dirección de memoria $02C6) de su computador.

Registro de Color 2
(Dirección de memoria $2C6)

Nº de bit	7	6	5	4	3	2	1	Ø
Contenido	1	Ø	Ø	1	1	Ø	Ø	*
	Color ($Ø9)				Lum. ($Ø4)			

* El bit 0 no se usa

Probando el Programa

Digite el programa y ejecútelo. Puede utilizar el comando "D" de su depurador MAC/65 o Atari para ver si funciona. Cuando escriba "D26C", échele un vistazo al contenido del Registro de Color 2. La línea en la pantalla debe mostrar que la dirección de memoria $2C6 (Registro de Color 2) contiene el valor $98. Convierta el número hexadecimal $98 a un número binario, y verá que es igual a 10011000, exactamente el mismo número binario indicado anteriormente en nuestro análisis bit por bit del Registro de Color 2. Esta misma técnica se puede utilizar también para cargar cualquier otro registro con cualquier otro color y luminancia en un programa de lenguaje ensamblador. Todo lo que debe hacer es sustituir algunas variables por números literales. He aquí una forma en que el programa podría ser reescrito para que sea más versátil:

Un programa mejor para configurar los colores

```
10   ;
20   ; PROGRAMA SETCLR
30   ;
40   CLRNR=$C0  ;NUMERO DE COLOR
50   HUENR=$C1  ;NUMERO DE MATIZ
60   CLREG=$2C6 ;NUMERO DE REGISTRO DE COLOR
70   ;
80     *=$0600
90   ;
100    LDA #$09 ;AZUL CLARO
110    STA CLRNR
120    LDA #$04 ;MATIZ #4
130    STA HUENR
```

```
140  ;
150    CLC ;LIMPIAR BANDERA DE ACARREO
160    CLD ;ACTIVAR BANDERA DECIMAL
170    LDA CLRNR
180    ASL A
190    ASL A
200    ASL A
210    ASL A
220    STA CLREG
230    LDA HUENR
240    ASL A
250    ADC CLREG
260    STA CLREG
270    .END
```

Este es en realidad dos programas en uno. De las líneas 100 a la 130, se pueden asignar valores en las variables que representan un color, una luminancia, y un registro de color. Luego, en las líneas de la 150 a la 260 del cuerpo del programa principal, se pueden cargar los valores de color y luminancia en cualquier registro de color. ¿Así que por qué no intentarlo? ¡Cambie las variables que se utilizan en las líneas 40 y 60, ejecute el programa varias veces, y vea cómo cambian los colores de su pantalla!

Una manera más fácil

También puede cambiar los colores de la pantalla sin tomarse la molestia de usar un montón de comandos ASL. Si lo desea, puede realizar todas las operaciones necesarias ASL en su cabeza, antes de escribir el programa. Por ejemplo, si usted multiplica por $ 10 (o 16 en decimal) el número de color Atari, obtendrá el mismo resultado que usted habría realizado con cuatro operaciones ASL sobre el número. Multiplique $09 por $10, y obtendrá $90, el mismo número que se obtendría mediante la utilización de cuatro operaciones ASL sobre el número $09. De manera similar, puede realizar una operación ASL sobre un número binario simplemente multiplicando por 2 (o, si lo prefiere, por $02). Lleve a cabo una operación ASL sobre el número $04 (0100 binario), y obtendrá $08 (1000 binario); el mismo número se obtendría si se multiplica $04 por 2. Si desea escribir un programa similar a SETCLR pero más fácil, puede hacerlo de la siguiente manera:

UN PROGRAMA MÁS FÁCIL QUE SETCLR

```
10   ;
20   ;UN PROGRAMA MÁS FACIL QUE SETCLR
30   ;
40   CLRNR=$C0 ;NUMERO DE COLOR
50   HUENR=$C1 ;NUMERO DE MATIZ
60   CLREG=$2C6 ;NUMERO DE REGISTRO DE COLOR
70   ;
80     *=$0600
90   ;
```

```
100    LDA #$90 ;COLOR No. 09 POR $10
110    STA CLRNR
120    LDA #$08 ;MATIZ No. 04 POR $2
130    STA HUENR
140  ;
150    CLC ;LIMPIAR BANDERA DE ACARREO
160    CLD ;LIMPIAR BANCERA DECIMAL
170    LDA CLRNR
180    STA CLREG
190    LDA HUENR
200    ADC CLREG
210    STA CLREG
220    .END
```

Al añadir un par de ciclos a un programa como este, además de un bucle infinito, es posible asignar a un registro de color un arco iris de colores en constante cambio. A continuación, puede hacer que el computador Atari dé vueltas una y otra vez en el ciclo, desplegando todos sus colores en la pantalla, sin parar, hasta que se presione la tecla BREAK o SYSTEM RESET, se apague el computador o se tire del enchufe. He aquí un programa que hará esto. Se ejecutará un ciclo sin fin pasando por todos los colores y combinaciones que puede generar su Atari, desplegando cada uno de ellos en el área del borde de la pantalla del computador. (Si el programa le parece familiar, es porque lo es. De acuerdo a lo prometido, corresponde a una versión en lenguaje ensamblador del programa de rotación de colores en BASIC que fue presentado en el capítulo 2.)



EL ARCOIRIS ATARI

```
10   ;
20   ;RAINBOW.SRC
30   ;
```

```
40   COLRBK=$2C8 ;EL REGISTRO DE COLOR DEL BORDE EN GRAPHICS 0
50   TMPCLR=$C0 ;UN LUGAR PARA ALMACENAR TEMPORALMENTE LOS COLORES
60   ;

70            *=$0600
80   ;
90   START    LDA #$FE ;VALOR DE COLOR MAXIMO
100           STA TMPCLR
110  ;
120  NEWCLR   LDA TMPCLR
130           STA COLRBK
140  ;
150           LDX #$FF
160  LOOPA    NOP ;SOLO UN CICLO DE PAUSA
170  ;
180           LDY #$30
190  LOOPB    NOP ;OTRO CICLO DE PAUSA
200           DEY ;DECREMENTAR REGISTRO Y
210           BNE LOOPB
220  ;
230           DEX ;DECREMENTAR REGISTRO X
240           BNE LOOPA
250  ;
260           DEC TMPCLR ;DECREMENTAR TMPCLR
270           DEC TMPCLR ;RESTAR 2 PARA OPTENER EL SIGUIENTE COLOR
280           BNE NEWCLR ;SI ES DISTINTO DE CERO, CAMBIE COLORES OTRA VEZ
290  ;
300           JMP START ;SE DESPLEGARAN TODO LOS COLORES – AHORA HAGAMOSLOS
                         TODOS DE NUEVO
```

LSR (Desplazamiento lógico a la derecha)

La instrucción LSR (Logical Shift Right - Desplazamiento Lógico a la derecha) es el opuesto exacto de la instrucción ASL, como puede observar en esta ilustración:

Una ilustración del mnemotécnico "LSR"

Acumulador

Ø → 7 6 5 4 3 2 1 Ø → C

Cómo Funciona la Instrucción LSR

LSR, tal como ASL, trabaja sobre cualquier número binario en el acumulador del 6502. Sin embargo, desplazará cada bit una posición a la derecha. Al bit 7 del nuevo número,

que ha quedado vacío producto de la instrucción LSR, se le asignará un cero. El LSB (Least Significant Bit - Bit menos significativo) será asignado a la bandera de acarreo del registro de P. Se puede utilizar la instrucción LSR para dividir por 2 cualquier número par de 8 bits, de la siguiente manera:

DIVIDIR UN NÚMERO POR 2 USANDO LA INSTRUCCION "LSR"

```
10   ;DIV2LSR.SRC
20   ;DIVIDIR POR 2 USADO LSR
30   ;
40   VALUE1=$C0
50   VALUE2=$C1
60   ;
70      *=$0600
80   ;
90      LDA #6
100     STA VALUE1
110  ;
120     LDA VALUE1
130     LSR A
140     STA VALUE2
150     .END
```

Esta rutina también puede ser utilizada para otro propósito. Si la ejecuta, y luego verifica la bandera de acarreo, podrá identificar si el número almacenado en VALUE1 es par o impar. Si la rutina deja el bit de acarreo desactivado, entonces el número que acaba de ser dividido es impar. ¡Si el bit de acarreo está activado, entonces el valor es par!

El siguiente es un programa que puede digitar, ejecutar y comprobar por medio del depurador para ver si un número es par o impar. Si el programa deja el número $FF en la dirección de memoria $C2, cuya etiqueta es FLGADR, entonces el número que fue dividido por 2 en la línea 160 es impar. Si el programa deja un 0 en la dirección FLGADR, entonces el número que se dividió es par:

```
10   ;ODDEVEN.SRC
20   ;PAR O IMPAR?
30   ;
40   VALUE1=$C0
50   VALUE2=$C1
60   FLGADR=$C2
70   ;
80      *=$0600
90   ;
100     LDA #7 ;(ODD)
110     STA VALUE1
120     LDA #0
130     STA FLGADR ;LIMPIAR FLGADR
140  ;
150     LDA VALUE1
```

```
160     LSR A ;EJECUTAR LA DIVISION
170     STA VALUE2 ;LISTO
180 ;
190     BCS FLAG
200     RTS ;TERMINE LA RUTINA SI SE LIMPIO EL ACARREO
210 ;
220     FLAG NOP
230     LDA #$FF ;DE LO CONTRARIO, ACTIVE FLAG
240     STA FLGADR
250     RTS  ;...Y TERMINE EL PROGRAMA
```

Descomprimiendo los datos

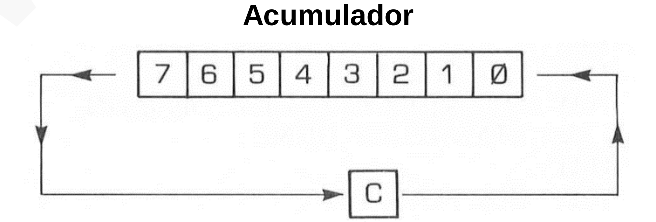
Como hemos mencionado, también puede utilizar LSR para descomprimir datos que han sido comprimidos utilizando ASL. Sin embargo, para descomprimir los datos, también hay que utilizar otro tipo de función en lenguaje ensamblador, llamada operador lógico. Hablaremos de los operadores lógicos y presentaremos una rutina de ejemplo para descomprimir datos más adelante en este capítulo. Mientras tanto, echemos un vistazo a dos operadores de desplazamiento de bits: ROL (que significa "ROtate Left - Rotar a la izquierda") y ROR (que significa "ROtate Right - Rotar a la derecha).

ROL (Rotar a la izquierda) y ROR (Rotar a la derecha)

Las instrucciones ROL (rotar a la izquierda) y ROR (rotar a la derecha) también se utilizan para desplazar bits en números binarios. Pero no hacen uso del bit de acarreo. En su lugar, trabajan de esta manera:

La Instrucción ROL ("Rotate Left - Rotar a la izquierda")

Acumulador



La Instrucción ROR ("Rotate Right - Rotar a la derecha")

Acumulador

Cómo funcionan "ROL" y "ROR"

Como puede ver, ROL y ROR funciona de manera similar a ASL y LSR, salvo que se asigna el bit de acarreo (en lugar de cero) al bit del extremo que quedó vacío producto de la rotación. ROL, tal como ASL, desplaza el contenido de un byte un lugar a la izquierda. Sin embargo, ROL no le asigna un cero al bit 0. En su lugar, asigna el bit de acarreo al bit 0 del número que está siendo desplazado, y que ha quedado vacío producto de la rotación, y le asigna el bit 7 al bit de acarreo. ROR funciona igual que ROL, pero en la dirección opuesta. Mueve cada bit de un byte una posición a la derecha, colocando el bit de acarreo en el bit 7 y el bit 0 en el bit de acarreo.

Los operadores lógicos

Antes de pasar a la aritmética binaria convencional, echemos un breve vistazo a cuatro mnemotécnicos importantes del lenguaje ensamblador llamados Operadores Lógicos. Estas instrucciones son AND ("y"), OR ("o"), EOR ("exclusive or - o exclusivo (disyunción exclusiva)"), y BIT ("bit"). Los cuatro operadores lógicos del 6502 parecen muy misteriosos a primera vista. Pero, típico del lenguaje ensamblador, pierden gran parte de su misterio una vez que se comprende cómo funcionan.

AND, ORA, EOR y BIT se usan para comparar valores. Pero trabajan de manera diferente a los operadores de comparación CMP, CPX y CPY. Las instrucciones CMP, CPX y CPY dan resultados muy generales. Todo lo que podemos determinar es si dos valores son iguales o no, y si son distintos, que uno es más grande que el otro. Las instrucciones AND, ORA, EOR y BIT son mucho más específicas. Están acostumbradas a comparar bits individuales de un número, y por lo tanto tienen todo tipo de usos.

Lógica Booleana

Los cuatro operadores lógicos del lenguaje ensamblador usan los principios de una ciencia matemática llamada Lógica Booleana. En la lógica booleana, los números binarios 0 y 1 no se utilizan para expresar valores, sino para indicar si una expresión es verdadera o falsa. Si una expresión es verdadera, se dice que su valor en la lógica booleana es 1. Si es falsa, se dice que su valor es 0. En el lenguaje ensamblador del 6502, el operador AND tiene el mismo significado que la palabra "AND" tiene en inglés. Si un bit y otro tienen el

mismo valor 1 (son "verdaderos"), entonces el operador también dará un valor 1. Pero si cualquier otra condición existe, si un bit es verdadero y el otra es falso, o si los dos bits son falsos, entonces el operador AND devuelve un resultado 0 (o falso).

Los resultados de los operadores lógicos a menudo se ilustran por medio de las denominadas "tablas de verdad". He aquí la tabla de verdad para el operador AND.

Tabla de verdad para AND

```
      0          0          1          1
AND   0    AND   1    AND   0    AND   1
-------    -------    -------    -------
      0          0          0          1
```

En el lenguaje ensamblador del 6502, la instrucción AND se utiliza a menudo en una operación llamada "bit masking - máscara de bits". El propósito de la máscara de bits es activar o desactivar bits específicos de un número. El operador AND se puede utilizar, por ejemplo, para desactivar cualquier número de bits poniendo un cero en cada bit que se desea borrar. Así es como funciona la máscara de bits:

```
100  LDA #$AA ; 10101010 BINARIO
110  AND #$F0 ; 11110000 BINARIO
```

Si su computador encuentra esta rutina en un programa, la siguiente operación AND se llevaría a cabo:

```
     1010 1010   (contenido del acumulador)
AND  1111 0000
--------------
     1010 0000   (nuevo valor del acumulador)
```

Como puede ver, esta operación desactiva el nibble bajo de $AA, dejándolo en $0 (dando un resultado de $A0). La misma técnica funciona con cualquier otro número 8 bits. No importa qué número pase a través de la máscara 1111 0000, su nibble bajo siempre saldrá sin cambios de la operación AND.

Descompresión de datos usando el operador "AND"

El operador AND, junto con la instrucción de desplazamiento de bits LSR, puede ser utilizada para descomprimir los datos que han sido comprimidos con la instrucción ASL. He aquí una rutina de ejemplo para descomprimir datos.

```
10   ;
20   ;DESEMPAQUETAR DATOS
```

```
30   ;
40   PKDBYT=$C0
50   LONYB=$C1
60   HINYB=$C2
70   ;
80      *=$0600
90   ;
100     LDA #$45 ;O CUALQUIER COSA
110     STA PKDBYT
120     LDA #0 ;LIMPIAR LONYB Y HINYB
130     STA LONYB
140     STA HINYB
150  ;
160     LDA PKDBYT
170     PHA ;GUARDARLO EN EL STACK
180     AND #$0F ;BINARIO 0000 1111
190     STA LONYB ;NIBBLE BAJO
200     PLA ;EXTRAER PKDBYT DEL STACK
210     LSR A
220     LSR A
230     LSR A
240     LSR A
250     STA HINYB ;NIBBLE ALTO
260     RTS
```

El Operador "ORA"

Cuando se utiliza la instrucción ORA ("o") para comparar un par de bits, el resultado de la comparación es 1 (verdadero) si el valor de cualquiera de los dos bits es 1. Esta es la tabla de verdad de ORA:

Tabla de verdad para ORA

```
      0        0        1        1
  ORA 0    ORA 1    ORA 0    ORA 1
  -----    -----    -----    -----
      0        0        1        1
```

ORA también se utiliza en las operaciones de máscara de bits. He aquí un ejemplo de una rutina de máscara de bits utilizando ORA:

```
LDA #VALUE
ORA $0F
STA DEST
```

Supongamos que el número en VALUE es $22 (binario 0010 0010). La siguiente es la operación de máscara de bits que tendría lugar.

```
      0010 0010    (en el acumulador)
ORA   0000 1111
---------------
      0010 1111    (nuevo valor en el acumulador)
```

El Operador "EOR"

La instrucción EOR ("o exclusivo", o disyunción exclusiva) devolverá un valor verdadero (1) si, y sólo si, uno de los dos bits es un 1. La siguiente es la tabla de verdad del operador EOR.

Tabla de verdad para EOR

```
      0         0         1         1
  EOR 0     EOR 1     EOR 0     EOR 1
  -----     -----     -----     -----
      0         0         1         0
```

La instrucción EOR se utiliza a menudo para comparar bytes y determinar si éstos son idénticos, ya que si cualquier bit de dos bytes es diferente, el resultado de la comparación será distinto de cero. He aquí una ilustración de esa comparación.

```
        Ejemplo 1                    Ejemplo 2

           1011 0110                     1011 0110
EOR        1011 0110       Pero : EOR    1011 0111
--------------------             ----------------
           0000 0000                     0000 0001
```

En el ejemplo 1, los bytes que se comparan son idénticos, por lo que el resultado de la comparación es cero. En el Ejemplo 2, un bit es diferente, por lo que el resultado de la comparación es distinto de cero. El operador EOR también se utiliza para obtener el complemento de un valor. Si se hace un EOR entre un valor de 8 bits y $FF, cada bit del valor que es un 1 será reemplazado por un 0, y cada bit que es un 0 será reemplazado por un 1.

```
      1110 0101    (en el acumulador)
EOR   1111 1111
---------------
      0001 1010    (nuevo valor en el acumulador)
```

Otra característica útil de la instrucción EOR es que cuando se ejecuta dos veces sobre un número con el mismo operando, el primer número será cambiado a otro número, y luego será de vuelto a su valor original. Esto se muestra en el ejemplo siguiente.

```
      1110 0101    (en el acumulador)
EOR   0101 0011
──────────────
      1011 0110    (nuevo valor en el acumulador)
EOR   0101 0011    (el mismo operando de arriba)
──────────────
      1110 0101    (se reestablece el valor original en el acumulador)
```

Esta capacidad de la instrucción EOR se utiliza a menudo en gráficos de alta resolución para poner una imagen sobre otra, sin destruir la de abajo. (¡Sí, así es como se hace!)

El operador "BIT"

Esto nos lleva al operador BIT, una instrucción más esotérica aún que AND, ORA, o EOR. La instrucción BIT se usa para determinar el estado de un bit - o bits – específico de un valor binario almacenado en la memoria. Cuando la instrucción BIT es utilizada en un programa, los bits 6 y 7 del valor examinado se transfieren directamente a los bits 6 y 7 del registro de estado del procesador (los bits de signo y de desbordamiento). Y entonces, se realiza una operación AND sobre el acumulador y el valor en la memoria. El resultado de esta operación AND se almacena en la bandera Z (cero) del registro P. Si hay un 1 tanto en el acumulador como en el valor en la memoria, en la misma posición del bit, el resultado no será cero y la bandera Z se desactivará. Si los bits son diferentes o los dos son igual a cero, el resultado será cero y la bandera Z se activará. El aspecto más importante aquí es que después de que todo esto ocurre, los valores en el acumulador y la posición de memoria permanecen sin cambios.

Capítulo Diez
Las matemáticas en el Lenguaje Ensamblador

Como programador de lenguaje ensamblador del Atari, es probable que nunca tenga que escribir un programa ultra sofisticado de aritmética con precisión arbitraria. Si alguna vez tiene que escribir un programa que incluya una gran cantidad de operaciones con precisión arbitraria, su Atari le puede ayudar con esto. Su computador tiene integradas en su sistema operativo un conjunto muy poderoso de programas de aritmética, llamados rutinas de punto flotante, o rutinas FP ("Floating Point - Punto Flotante"). La gente de Atari ha tenido el cuidado de ofrecerle la posibilidad de utilizar estas rutinas del OS ("Operating System - Sistema Operativo") en sus programas de lenguaje ensamblador. Se han publicado instrucciones acerca de cómo usar el paquete FP de Atari en un número de publicaciones, entre ellas en el De Re Atari, que es un manual publicado por Atari para los programadores en lenguaje ensamblador.

Incluso si no desea utilizar el paquete de rutinas FP incorporado en su Atari (y hay razones para no hacerlo, las rutinas son lentas), puede encontrar código que ya sido escrito para la mayoría de las operaciones aritméticas sofisticadas (a menudo llamadas operaciones binarias de precisión arbitraria) en una serie de manuales sobre programación en lenguaje ensamblador del 6502. Un texto lleno de programas de precisión arbitraria que puede digitar y usar en su computador es el libro "6502 Assembly Language Subroutines", publicado por Osborne / McGraw Hill y escrito por Lance A. Leventhal y Winthrop Saville.

¿Entonces para qué molestarse?

Entonces se puede preguntar para qué tomarse la molestia de incluir un capítulo en este libro acerca de la aritmética avanzada del 6502. La respuesta es que no importa cuánta ayuda se encuentre disponible, si quiere ser un buen programador en lenguaje ensamblador, tiene que conocer los principios de la aritmética avanzada del 6502. Así que, aunque puede que nunca tenga que escribir una rutina en lenguaje ensamblador que calcule una división larga con números con signo y una precisión de 17 decimales, es muy probable que tarde o temprano tengamos que utilizar algunas de estas operaciones aritméticas en sus programas.

De vez en cuando la mayoría de los programadores en lenguaje ensamblador tiene que escribir una rutina de suma o resta, o una rutina para multiplicar o dividir un par de números, o un programa que trabaje con números con signos o números BCD ("Binary-Coded Decimal" - "decimal codificado en binario"). Las operaciones lógicas que son ampliamente utilizadas en los programas del 6502 también caen bajo la categoría de matemáticas en el lenguaje ensamblador. En este capítulo, por lo tanto, revisaremos la suma, resta y multiplicación binaria de 8 y 16 bits. También se discutirá un poco acerca

de la división larga binaria. Vamos a cerrar el capítulo con una breve introducción a los números con signo y al sistema numérico BCD.

En el problema de sumas que invocó desde el BASIC en el capítulo 8, pudo ver cómo trabaja el bit de acarreo del registro de estados del procesador en operaciones de sumas de 16 de bits. Ahora vamos a revisar el uso del bit de acarreo en problemas de suma, y también veremos cómo funciona el acarreo en problemas de resta, multiplicación y división.

Una mirada de cerca al bit de acarreo

La mejor manera de ver cómo funciona el bit de acarreo es mirándolo de cerca a través de un microscopio "electrónico", a nivel de bit. Analicemos estos dos simples problemas de suma de 4 bits en notación hexadecimal y binaria, y verá claramente cómo ninguna operación de suma genera un acarreo, tanto en notación binaria como en hexadecimal.

```
   HEXADECIMAL      BINARIO

       04             0100
   +   01         +   0001
       ——             ————
       05             0101


       08             1000
   +   03         +   0011
       ——             ————
       0B             1011
```

Ahora echemos un vistazo a un par de problemas que usan números más grandes (de 8 bits). El primero de estos dos problemas no genera un acarreo, pero el segundo sí.

```
   HEXADECIMAL      BINARIO

      8E              1000 1110
   +  23         +    0010 0011
      ——              —————————
      B1              1011 0001


      8D              1000 1101
   +  FF         +    1111 1111
      ——              —————————
      18C        (1)  1000 1100
```

Fíjese que la suma en el segundo problema corresponde a un número de 9 bits: 1 1000 1100 en binario, o 18C en notación hexadecimal. He aquí un programa en lenguaje ensamblador que llevará a cabo exactamente el mismo problema de sumas. Digítelo, ejecútelo, y podrá ver cómo funciona la bandera de acarreo de su computador:

Suma de 8 bits con acarreo

```
10     *=$0600
20     CLD
30     CLC
40     LDA #$80
50     ADC #$FF
60     STA $CB
70     RTS
```

Cuando haya digitado este programa, ensámblelo y ejecútelo usando el comando "G" del depurador de su ensamblador. Cuando el programa se haya ejecutado, y mientras el depurador se encuentre funcionando, escriba el comando "DCB" ("Display memory location $CB - Despliegue la posición de memoria $CB"). A continuación verá en su pantalla de video una línea como esta:

```
00CB 8C 00 00 00 00
```

Esa línea nos muestra que la dirección de memoria $CB ahora contiene el número $8C, la suma correcta de los números que sumamos, excepto por el acarreo. Así que ¿dónde está el acarreo? Bueno. Si lo que ha leído en este libro acerca del bit de acarreo es correcto, debería estar en el bit de acarreo del registro P de su computador. De la manera en que nuestro programa está escrito ahora, no hay una manera fácil de averiguar si el bit de acarreo de nuestra operación ha sido asignado al bit de acarreo del registro P. Pero al agregar un par de líneas a nuestro programa, y al ejecutarlo nuevamente, lo podremos descubrir. He aquí cómo se debe reescribir el programa para que podamos verificar el bit de acarreo:

```
10      *=$0600
20      CLD
30      CLC
40      LDA #$8D
50      ADC #$FF
60      PHP
70      STA $CB
80      PLA
90      AND #01
100     STA $CC
110     RTS
```

En esta reescritura de nuestro programa original, hemos utilizado una nueva instrucción de manipulación de la pila: PHP. PHP significa "PusH Processor status (P register) on

stack - Ponga el registro de estado del procesador (P) en la pila". También hemos utilizado el operador AND que fue presentado en el capítulo 9. Además, hemos utilizado una instrucción de manipulación de la pila que vimos hace un par de capítulos: PLA, que significa "PulL contents of Accumulator from stack - Extrae el contenido del acumulador desde la pila".

La instrucción PHP se usa en la línea 60 de nuestro programa reescrito. Está allí porque queremos guardar el contenido del registro P tan pronto como los números #$8D y #$FF hayan sido sumados. Podemos utilizar la instrucción PHP sin ningún temor de que le haga algo terrible a nuestro programa, ya que es una instrucción que no afecta el contenido del registro P ni del acumulador, pero sí afecta al puntero de la pila ("Stack pointer").

Al ejecutar este programa, lo primero que se hará es sumar los números literales $8D y $FF. Sin embargo, antes de que se almacene el resultado de este cálculo en alguna parte, se guarda el contenido del registro de estado en la pila, utilizando la instrucción PHP. Cuando esta operación se complete, el valor del acumulador (todavía la suma de $8D y $FF, sin acarreo) se almacenará en la dirección de memoria $CB. A continuación, en la línea 80, cuando ya casi se han ejecutado todas las órdenes, se elimina el valor que fue puesto en la pila por la instrucción PHP de la línea 60. Entonces, puesto que la única bandera del Registro P que nos interesa es la bandera de acarreo (bit 0), hemos utilizado una instrucción AND para enmascarar cada bit, excepto el bit 0, del número que acaba de ser extraído de la pila. Por último, el número resultante - que debería ser $01 si nuestras instrucciones hasta ahora se han ejecutado correctamente - se almacena en la dirección de memoria $CC.

Ahora, cuando queramos, podremos mirar en la dirección de memoria y ver cuál fue el resultado (sin acarreo) del cálculo en el programa. Entonces podremos mirar en la dirección de memoria $CC y echar un vistazo al estado del registro P justo después de haber sumado los números $8D y $FF. ¡Así que hagámoslo! Ensamble el programa, ejecútelo utilizando el comando "G" de su depurador, y luego use el comando "DCB" a echar un vistazo al contenido de la dirección de memoria $CB y las direcciones que vienen a continuación. Esto es lo que verá:

```
00CB 8C 01 00 00 00
```

Esa línea nos dice dos cosas: que la dirección de memoria $CB contiene el número $8C, el resultado de nuestro cálculo, sin acarreo, y que la suma de #$8D y #$FF efectivamente activó el bit de acarreo del registro de estado del procesador.

Suma de 16 Bits

Ahora echaremos un vistazo a un programa que suma dos números de 16 bits. Los mismos principios que se utilizan en este programa también pueden ser usados para escribir programas que sumen números de 24 bits, 32 bits, y más. Aquí está el programa:

UN PROGRAMA DE SUMAS CON PRECISION ARBITRARIA

```
10    ;
20    ; ESTE PROGRAMA SUMA UN NUMERO DE 16 BITS ALMACENADO EN $B0 Y
        $B1
30    ; A OTRO NUMERO DE 16 BITS ALMACENADO EN $C2 Y $C3
40    ; Y GUARDA EL RESULTADO EN $C2 Y $C3
50    ;
60      *=$0600
65    ;
70      CLD
80      CLC
90      LDA $B0 ;PARTE BAJA DEL NUMERO DE 16 BITS ALMACENADO EN $B0
        Y $B1
100     ADC $C0 ;PARTE BAJA DEL NUMERO DE 16 BITS ALMACENADO EN $C0
        Y $C1
110     STA $C2
120     LDA $B1 ;PARTE ALTA DEL NUMERO DE 16 BITS ALMACENADO EN $B0
        Y $B1
130     ADC $C1 ;PARTE ALTA DEL NUMERO DE 16 BITS ALMACENADO EN $C0
        Y $C1
140     STA $C3
150     RTS
```

Cuando vea este programa, recuerde que su computador Atari almacena los números de 16 bits en orden inverso - el byte alto en la segunda dirección, y el byte bajo en la primera dirección. Una vez que entienda este enredo, verá que la suma binaria de 16 bits no es difícil de comprender. En este programa, lo primero que se hace es desactivar la bandera de acarreo del registro P. Luego sumamos el byte bajo del número de 16 bits que se encuentra en las direcciones $BO y $B1 con el byte bajo del número de 16 bits que se encuentra en las direcciones $CO y $C1. El resultado de este cálculo se coloca entonces en la dirección de memoria $C2. Si hay un acarreo, el bit de acarreo del registro P se activará automáticamente.

En la segunda parte del programa, el byte alto del número que se encuentra en las direcciones $BO y $B1 se suma al byte alto del número que se encuentra en las direcciones $CO y $C1. Si como resultado de la suma anterior se ha activado el bit de acarreo del registro P, entonces también se sumará un acarreo a los bytes altos de los dos números que se suman. Entonces el resultado de esta parte de nuestro programa será almacenado en la dirección de memoria $C3. Cuando esta operación se haya completado, los resultados de nuestra suma serán almacenados, comenzando por el byte bajo, en las direcciones de memoria $C2 y $C3.

Resta de 16 Bits

He aquí un programa de restas de 16 bits:

```
10    ;
20    ; ESTE PROGRAMA RESTA EL NUMERO DE 16 BITS ALMACENADO EN $B0
        Y $B1
30    ; CON EL NUMERO DE 16 BITS ALMACENADO EN $C0 Y $C1
40    ; Y GUARDA EL RESULTADO EN $C2 Y $C3
50    ;
60      *=$0600
65    ;
70      CLD
80      SEC ;ACTIVA BANDERA ACARREO
90      LDA $00 ;PARTE BAJA DEL NUMERO DE 16 BITS ALMACENADO EN $C0
        Y $C1
100     SBC $B0 ;PARTE BAJA DEL NUMERO DE 16 BITS ALMACENADO $B0 Y
        $B1
110     STA $C2
120     LDA $C1 ;PARTE ALTA DEL NUMERO DE 16 BITS ALMACENADO EN $C0
        Y $C1
130     SBC $B1 ;PARTE ALTA DEL NUMERO DE 16 BITS ALMACENADO EN $B0
        Y $B1
140     STA $C3
150     RTS
```

Dado que la resta es el opuesto exacto de la suma, la bandera de acarreo tendrá que ser activada (y no desactivada) antes de que se lleve a cabo una resta en aritmética binaria del 6502. En la resta, la bandera de acarreo se trata como un préstamo, no como un acarreo, y por lo tanto, se debe activar, y no desactivar, de modo que si es necesario un préstamo, haya un valor al cual pedirle ese préstamo. Luego de activar el bit de acarreo, hacer una resta en el 6502 es sencillo. En nuestro problema de ejemplo, al número de 16 bits que se encuentra en las direcciones $C0 y $C1 se le resta, comenzando por el byte bajo, el número que se encuentra en las direcciones $B0 y $B1. El resultado de nuestra resta (incluido el préstamo del byte alto, si es que fue necesario hacer un préstamo) se almacena en las direcciones de memoria $C2 y $C3.

Multiplicación binaria

No hay instrucciones en lenguaje ensamblador del 6502 para la multiplicación ni división. Para multiplicar un par de números por medio del lenguaje ensamblador del 6502, tiene que realizar una serie de sumas. Para dividir números, tiene que realizar una serie de restas. He aquí un ejemplo de cómo dos números binarios de 4 bits se pueden multiplicar usando los principios de la suma:

```
    0110 ($06)
X   0101 ($05)
--------------
    0110
   0000
  0110
 0000
--------------
0011110 ($1E)
```

Observe lo que ocurre cuando se trabaja en esta multiplicación. En primer lugar, 0110 es multiplicado por 1. Se escribe el resultado de esta operación, el que también es 0110.

Lo que sucede a continuación

A continuación, 0110 es multiplicado por 0. El resultado de esta operación, una cadena de ceros, se escribe un espacio desplazado a la izquierda. Luego 0110 es multiplicado por 1 otra vez, y este resultado se escribe, una vez más, desplazado a la izquierda. Por último, otra multiplicación por cero da como resultado otra cadena de ceros, que también se escribe desplazada un espacio a la izquierda. Una vez que se ha hecho esto, todos los productos parciales de nuestras multiplicaciones se suman, de la misma manera que si se tratara de una multiplicación convencional. El resultado de esta suma, como se puede ver, es el producto final $1E.

Esta técnica de multiplicación funciona bien, pero en realidad es bastante arbitraria. ¿Por qué, por ejemplo, desplazamos a la izquierda cada producto parcial en este problema antes de escribirlo? Podríamos haber logrado el mismo resultado desplazando el producto parcial arriba a la derecha antes de sumar. Es exactamente lo que se hace a menudo en la multiplicación del 6502: en lugar de desplazar cada producto parcial a la izquierda antes de guardarlo en la memoria, muchos de los algoritmos de multiplicación del 6502 desplazan el producto parcial anterior a la izquierda antes de sumarlo al nuevo.

Multiplicación de precisión arbitraria

A continuación le presentamos un programa que le mostrará cómo funciona.

UN PROGRAMA PARA MULTIPLICAR CON PRECISION MULTIPLE

```
10    MPR=$C0 ;MULTIPLICADOR
20    MPD1=$C1 ;MULTIPLICANDO
30    MPD2=$C2 ;NUEVO MULTIPLICANDO DESPUES DE 8 DESPLAZAMIENTOS
40    PRODL=$C3 ;BYTE BAJO DEL PRODUCTO
50    PRODH=$C4 ;BYTE ALTO DEL PRODUCTO
60    ;
70           *=$0600
```

```
80    ;
85    ;       ESTOS SON LOS NUMEROS QUE MULTIPLICAREMOS
87    ;
90            LDA #250
100           STA MPR
110           LDA #2
120           STA MPD1
130   ;
140   MULT    CLD
150           CLC
160           LDA #0 ;LIMPIA ACUMULADOR
170           STA MPD2 ;LIMPIA DIRECCION DEL MULTICANDO DESPLAZADO
180           STA PRODL ;LIMPIA BYTE BAJO DE LA DIRECCION DEL
              PRODUCTO
190           STA PRODH ;LIMPIA BYTE ALTO DE LA DIRECCION DEL
              PRODUCTO
200           LDX #8 ;USAREMOS EL REGISTRO X COMO CONTADOR
210   LOOP    LSR MPR ;DESPLAZA EL MULTIPLICADOR A LA DERECHA; EL
              BIT MENOS SIGNIFICATIVO CAE EN EL BIT DE ACARREO
220           BCC NOADD ;PRUEBA EL BIT DE ACARREO; SI ES CERO,
              SALTE A NOADD
230           CLC
240           LDA PRODL
250           ADC MPD1 ;SUMA EL BYTE BAJO DEL PRODUCTO AL
              MULTIPLICANDO
260           STA PRODL ;EL RESULTADO ES EL NUEVO BYTE BAJO DEL
              PRODUCTO
270           LDA PRODH ;CARGA EL ACUMULADOR CON EL BYTE ALTO DEL
              PRODUCTO
280           ADC MPD2 ;SUMA LA PARTE ALTA DEL MULTIPLICANDO
290           STA PRODH ;EL RESULTADO ES EL NUEVO BYTE ALTO DEL
              PRODUCTO
300   NOADD   ASL MPD1 ;DESPLAZA EL MULTIPLICANDO A LA IZQUIERDA;
              EL BIT 7 CAE EN EL ACARREO
310           ROL MPD2 ;ROTAR EL BIT DE ACARREO EN EL BIT 7 DE MPD2
320           DEX ;DECREMENTA EL CONTENIDO DEL REGISTRO X
330           BNE LOOP ;SI EL RESULTADO NO ES CERO, SALTE A LOOP
340           RTS
350           .END
```


Un procedimiento complejo

Como puede ver, la multiplicación binaria de 8 bits no es exactamente instantánea. Involucra un montón de desplazamientos de bits a la izquierda y a la derecha, y es difícil seguirles la pista. En el programa anterior, la manipulación más difícil de seguir es, probablemente, la relacionada con el multiplicando (MPD1 y MPD2). El multiplicando es sólo un valor de 8 bits, pero es tratado como un valor de 16 bits, porque es desplazado hacia la izquierda todo el tiempo, y mientras se está moviendo, se necesita una dirección de 16 bits para almacenarlo (en realidad, dos direcciones de 8 bits).

Para que vea usted mismo cómo funciona el programa, digítelo y ensámblelo. Luego, utilice el comando "G" de su depurador para ejecutarlo. Entonces, mientras usted todavía está en el modo de depuración, escriba "DC3" ("Display $C3"), y eche un vistazo a los contenidos de las direcciones de memoria $C3 y $C4, que deben contener el producto de 16 bits entre el número decimal 2 y el número decimal 250, que el programa se supone multiplica. El valor en las direcciones $C3 y $C4 debe ser $01F4 (desplegándose primero el byte bajo), el equivalente hexadecimal del número 500 decimal, o sea, el producto correcto.

Este no es el programa de multiplicación definitivo

Aunque el programa que acabamos de describir funciona bien, hay muchos otros algoritmos de multiplicación binaria, y algunos de ellos son más cortos y más eficientes que el que se acaba de presentar. El siguiente programa, por ejemplo, es mucho más corto que nuestro primer ejemplo, y por lo tanto más eficiente con la memoria y se ejecuta más rápido. Uno de sus trucos más bonitos es que utiliza el acumulador del 6502, en lugar de una dirección de memoria, para el almacenamiento temporal de los resultados del problema.

UN PROGRAMA PARA MULTIPLICAR MEJORADO

```
10    ;
20    PRODL=$C0
25    PRODH=$C1
30    MPR=$C2
40    MPD=$C3
50    ;
60              *=$0600
70    ;
80    VALUES    LDA #10
90              STA MPR
100             LDA #10
110             STA MPD
120   ;
130             LDA #0
140             STA PRODL
150             LDX #8
160   LOOP      LSR MPR
170             BCC NOADD
180             CLC
190             ADC MPD
200   NOADD     ROR A
210             ROR PRODL
220             DEX
230             BNE LOOP
235             STA PRODH
240             RTS
250             .END
```

Otra Prueba

Si lo desea, puede probar este programa de multiplicaciones mejorado de la misma manera en que lo probó anteriormente: ejecutándolo por medio del comando "G" y, a continuación, echando un vistazo a los resultados usando el comando "D".

Un comando diferente

Esta vez debe digitar "DC0", ya que el resultado de la multiplicación se almacena en las direcciones $C0 y $C1. El valor de 16-bits en $C0 y $C1 debería ser $0064 (con el byte más bajo primero), el equivalente hexadecimal del número 100 decimal, y que corresponde a la solución de este problema de multiplicación.

Siéntase libre de jugar

Puede jugar como quiera con estos dos problemas de multiplicación, probando diferentes valores e incluso llamando a estos programas desde el BASIC, de la forma en que lo hicimos con nuestro problema de sumas de 16 bits que vimos hace unos capítulos. Sin embargo, la mejor manera de familiarizarse con cómo funciona la multiplicación binaria, es teniendo algunos problemas a mano, y utilizar las herramientas que usaban nuestros antepasados: un lápiz y un trozo de papel. Solucione suficientes problemas de multiplicación binaria en papel, y pronto comenzará a entender los principios de la multiplicación del 6502.

División binaria con precisión arbitraria

Es poco probable que alguna vez tenga la oportunidad de escribir un programa de división larga binaria de precisión arbitraria. Y aun cuando la necesidad surja, probablemente no usará el programa limitado que explicamos aquí.

Sin embargo...

Este capítulo no estaría completo sin un ejemplo de un programa de división de números binarios largos. Así que aquí está, un simple (pero difícil) programa para dividir un dividendo de 16 bits por un divisor de 8 bits. El resultado es un cuociente de 8 bits.

Un programa truculento

Este programa ha sido diseñado de manera más sutil que el programa de multiplicación que presentamos algunos párrafos atrás. Durante la ejecución del programa, la parte alta del dividendo se almacena en el acumulador, y la parte baja del dividendo se almacena en una variable llamada DVDL. El programa contiene varios desplazamientos ("shifting"), rotaciones, restas, y decrementos sobre el registro X. Cuando termina su ejecución, el

cuociente queda en una variable llamada QUOT y el resto en el acumulador. Esto es correcto hasta que en la línea 380 el resto es sacado del acumulador y almacenado en la variable llamada RMDR. Finalmente, se termina el programa con una instrucción RTS.

UN PROGRAMA DE DIVISION SIMPLE

```
10    ;
20    ; DIVISION.SRC
30    ;
40            *=$0600
50    ;
60    DVDL=$C0 ;PARTE BAJA DEL DIVIDENDO
70    DVDH=$C1 ;PARTE ALTA DEL DIVIDENDO
80    QUOT=$C2 ;CUOCIENTE
90    DIVS=$C3 ;DIVISOR
100   RMDR=$C4 ;RESTO
110   ;
120           LDA #$1C ;SOLO UN VALOR DE EJEMPLO
130           STA DVDL
140           LDA #$02 ;EL DIVIDENDO AHORA ES $021C
150           STA DVDH
160           LDA #$05 ;OTRO VALOR DE EJEMPLO
170           STA DIVS ;ESTAMOS DIVIDIENDO POR 5
180   ;
190           LDA DVDH ;EL ACUMULADOR ALMACENARA DVDH
200           LDX #08 ;PARA UN DIVISOR DE 8 BITS
210           SEC
220           SBC DIVS
230   DLOOP   PHP ;EL CICLO QUE DIVIDE
240           ROL QUOT
250           ASL DVDL
260           ROL A
270           PLP
280           BCC ADOIT
290           SBC DIVS
300           JMP NEXT
310   ADDIT   ADC DIVS
320           NEXT DEX
330           BNE DLOOP
340           BCS FINI
350           ADC DIVS
360           CLC
370   FINI    ROL QUOT
380           STA RMDR
390           RTS ;ENDIT
```

No es un programa de división definitivo

A pesar de lo complejo que se ve este programa, no es de ninguna manera la mejor rutina de división de números binarios largos del mundo. No es el programa de división más preciso que verá jamás, y tampoco sirve para dividir números con decimales, fracciones, números muy largos, o números con signo. Si necesita un programa versátil de divisiones con precisión múltiple exacta, tendrá que echarle una mirada al paquete de punto flotante que viene integrado en el sistema operativo de su Atari.
El paquete de rutinas de punto flotante de Atari no es fácil de usar. Puede encontrar instrucciones más o menos completas para utilizarlo en la guía del programador De Re Atari. Si decide no utilizar el paquete FP de su computador, puedes echarle un vistazo a muchas rutinas de división y aritméticas que vienen incluidos en muchos "recetarios (cookbooks)" y manuales del lenguaje ensamblador del 6502. Otras rutinas aritméticas se encuentran publicadas en manuales como el excelente texto "6502 assembly language subroutines" por Leventhal y Saville (Berkeley: Osborne / McGraw-Hill, 1982).

Números con signo

Antes de pasar al siguiente capítulo, hay dos temas que debemos cubrir brevemente: los números con signo y los números BCD (decimal codificado en binario). Primero vamos a hablar de los números con signo. Las operaciones aritméticas no se pueden realizar sobre los números con signo utilizando las técnicas que se han descrito hasta ahora en este capítulo. Sin embargo, si se realizan ligeras modificaciones a estas técnicas, el chip 6502 de su computador Atari será capaz de sumar, restar, multiplicar y dividir números con signo. Si desea realizar operaciones aritméticas con números con signo, lo primero que tiene que saber es cómo representar los signos. Afortunadamente, eso no es difícil. Para representar un número con signo en la aritmética binaria, lo único que tiene que hacer es acordar que el bit de más a la izquierda (bit 7) representará al signo positivo o negativo. En la aritmética binaria con signos, si el bit 7 de un número es cero, entonces se trata de un número positivo. Si el bit 7 es un 1, entonces se trata de un número negativo.

Obviamente, si usted reserva un bit de un número de 8 bits para representar su signo, ya no se tiene un número de 8 bits. Lo que se tiene es un número de 7 bits o, si lo desea expresar de otro modo, tiene un número con signo que puede representar los valores desde el -128 al 127 en lugar del 0 al 255. También debería ser obvio que se necesita algo más que acordar que un bit representará el signo de un número para poder cambiar de operaciones de aritméticas binaria sin signo a operaciones de aritmética binaria con signo. Consideremos, por ejemplo, lo que obtendríamos si tratamos de sumar los números 5 y -4 y sólo consideramos usar el bit 7 para guardar el signo del número:

```
     0000 0101  (+5)
+    1000 0100  (-4)
     ---------
     1000 1001  (-9)
```

Esta respuesta es incorrecta. La respuesta correcta es 1. La razón por la que llegamos a la respuesta equivocada es porque tratamos de resolver el problema sin necesidad de utilizar un concepto que es fundamental para el uso de aritmética binaria con signos: el concepto de complementos.

Los complementos se utilizan en la aritmética binaria con signos porque los números negativos son complementos de los números positivos. Y los complementos de los números son muy fáciles de calcular en la aritmética binaria. En matemáticas binarias, el complemento de un 0 es 1, y el complemento de un 1 es 0. Se podría suponer, por tanto, que el complemento negativo de un número binario positivo se podría calcular complementando cada 0 en un 1, y cada 1 en un 0 (Por supuesto, con excepción del bit 7, que debe ser utilizado para representar el signo del número). Esta técnica de cálculo del complemento de un número intercambiando los bits 0 por 1 y los bits 1 por 0 tiene un nombre en el área de los lenguajes ensambladores. Se llama complemento a uno.

Para ver si la técnica del complemento a uno funciona, vamos a intentar usarlo para sumar dos números con signo, por ejemplo +8 y -5.

```
     0000 1000  (+8)
+    1111 1010  (-5)  (complemento a uno)
     ---------
     0000 0010  (+2)  (más acarreo)
```

¡Ups! ¡Eso también está mal! La respuesta debe ser +3. Bueno, eso nos lleva de nuevo a la mesa de diseño. La aritmética del complemento a uno no funciona.

Pero hay otra técnica, que está muy cerca de complemento a uno, que funciona. Se llama complemento a dos, y funciona de la siguiente manera: en primer lugar se calcula el complemento a uno de un número positivo. Después, simplemente sume 1. Eso le dará el complemento a dos, el verdadero complemento del número. Luego puede utilizar las reglas convencionales de las matemáticas binarias con números con signo. Y si no comete errores, funcionarán todo el tiempo. He aquí cómo funciona:

```
     0000 0101  (+5)
+    1111 1000  (-8)  (complemento a dos)
     ---------
     1111 1101  (-3)
```

He aquí otro problema de sumas con complemento a dos

```
     1111 1011  (-5)  (complemento a dos)
 +   0000 1000  (+8)
    ___________
     0000 0011  (+3)  (más acarreo)
```

Como hemos dicho, funciona todo el tiempo. Desafortunadamente, no es fácil explicar por qué. Existen algunas bonitas comprobaciones matemáticas, y si está interesado puede encontrarlas en numerosos libros sobre teoría de números binarios. Por el momento, lo más importante acerca de la aritmética del complemento a dos es saber cómo usarla, en caso de que tengamos que hacerlo.

Utilizando la bandera de desbordamiento

Hay otro hecho importante que debe recordar acerca de la aritmética binaria con signos: cuando se suman números con signo, para llevar el acarreo de un byte a otro se utiliza la bandera de desbordamiento (V) en lugar de la bandera de acarreo. La razón es la siguiente: La bandera de acarreo del registro P se activa cuando hay un desbordamiento a partir del bit 7 del número binario. Pero cuando este número es un número con signo, el bit 7 es el bit de signo - ¡No forma parte del número en sí! Así que la bandera de acarreo no se puede utilizar para detectar un acarreo en una operación que trabaja con números con signo. Puede resolver este problema utilizando el bit de desbordamiento del registro de estado del procesador. El bit de desbordamiento se activa cuando hay un desbordamiento a partir del bit 6, no del bit 7. Así que puede ser utilizado como un bit de acarreo en las operaciones aritméticas sobre números con signo.

Números BCD ("Binary Coded Decimal - Decimal Codificado en Binario")

Otra variedad de la aritmética binaria de la cual es útil saber algo es el sistema BCD (decimal codificado en binario). En la notación BCD se expresan los dígitos del 0 al 9 tal como se hace en la notación binaria convencional, pero los dígitos hexadecimales de la A a la F (del 1010 al 1111 en binario) no se utilizan. Por lo tanto, los números largos deben ser representados en notación BCD de una manera distinta a la notación binaria convencional. El número decimal 1258, por ejemplo, se escribe en notación BCD de la siguiente manera:

```
        1          2          5          8
0000 0001  0000 0010  0000 0101  0000 1000
```

En notación binaria convencional, el mismo número se puede escribir como:

```
 $0   $4   $E   $A
0000 0100 1110 1010
```

Esto equivale a $04EA, o a su equivalente hexadecimal 1258. La notación BCD se utiliza a menudo en programas de contabilidad, porque la aritmética BCD, a diferencia de la aritmética binaria directa, es 100% exacta. Los números BCD a veces se usan cuando se desea imprimirlos inmediatamente, cifra por cifra, a medida que son utilizados - por ejemplo, cuando los números registran por pantalla el puntaje de un juego.

La principal desventaja de los números BCD es que es difícil trabajar con ellos. Cuando se utilizan números BCD, se debe ser extremadamente cuidadoso con los signos, los puntos decimales y las operaciones de acarreo, o si no, puede terminar en un caos. También debe decidir si desea utilizar un byte de 8 bits para cada dígito (Lo que desperdicia memoria, ya que en realidad sólo toma 4 bits el codificar un dígito BCD) o si desea "empaquetar" dos dígitos en cada byte, lo que ahorra memoria pero consume tiempo de procesamiento.

Afortunadamente, como hemos señalado, probablemente nunca tendrá que utilizar la mayoría de las técnicas de programación que se describen en este capítulo, pero una comprensión acerca de cómo funcionan sin duda le harán un mejor programador en lenguaje ensamblador del Atari.

Capítulo Once
Más allá de la Página 6

La mayoría de los programadores principiantes en lenguaje ensamblador escriben rutinas breves que caben sin problemas en bloques cortos de memoria. Es por eso que los ingenieros que diseñaron su Atari reservaron la página 6, el bloque de memoria que va desde la dirección $0600 a la $06FF, para almacenar los programas en lenguaje ensamblador escritos por el usuario. Sin embargo, a medida que usted se va haciendo más y más experto en programación de lenguaje ensamblador, es muy probable que a la larga empiece a escribir programas que consumen más de 256 bytes de memoria, que son los que tiene la página 6. El averiguan en qué lugar de la memoria de su computador Atari se deben poner los programas largos de lenguaje ensamblador puede llegar a ser un problema difícil de solucionar.

El problema principal normalmente no es la cantidad de memoria libre que se encuentra disponible, sino más bien en qué lugar de la memoria RAM de su computador se encuentra. En la mayoría de los computadores, casi toda la memoria disponible para ser usada en programas escritos por los usuarios está en un solo lugar. Por lo general, se extiende a partir de una dirección baja, justo por encima de donde termina el sistema operativo de su computador, hasta llegar a una dirección alta, justo debajo del lugar donde comienza el bloque de memoria RAM denominado "memoria de la pantalla".

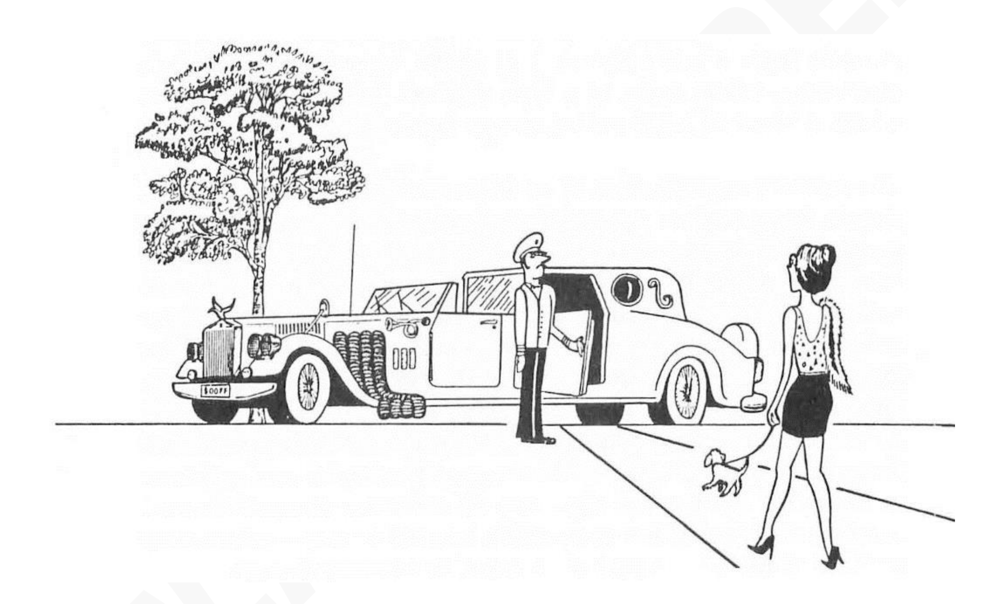
La organización de la memoria de su computador Atari no es tan simple. El espacio disponible para los programas escritos por el usuario se encuentra disperso por todo el mapa de memoria, y aprender a descubrir esos pequeños rincones en donde se puede almacenar código objeto sin dañar el sistema operativo de su Atari puede convertirse en todo un reto, y a veces, en una tarea frustrante. Esta situación se da porque hay diferentes tipos de computadores Atari, y porque todos son máquinas muy versátiles. Los computadores Atari tienen una capacidad de memoria RAM que va desde los 16K a los 64K, y pueden ser usados en diferentes modos gráficos y con una o hasta cuatro unidades de disco. Sin embargo, todos son compatibles desde el punto de vista del software, y mantenerlos de esta manera ha llevado a algunos interesantes trucos que Atari ha realizado en el diseño de su memoria.

Desafortunadamente, la gente que diseñó los computadores Atari todavía no ha encontrado una manera - y es dudoso que alguien pueda - para mantener compatibles todos los computadores Atari y al mismo tiempo mantener simple la organización de la memoria. Sin embargo, existen algunos lugares seguros en la memoria de su Atari donde puede almacenar sus programas en lenguaje ensamblador. Para ayudarle a encontrarlos, aquí tiene un mapa de memoria simple de la RAM de su computador:

El Distrito de los alquileres elevados

Página Cero
Desde $0000 a $00FF

La página cero, el bloque de memoria que se extiende desde $0000 a $00FF, corresponde al "distrito de los alquileres elevados" en la memoria RAM de su computador. El espacio de memoria acá es tan valioso que muy poco de éste se encuentra disponible para su uso a corto plazo por parte de los programas escritos por el usuario. Si desea escribir programas de alto rendimiento, en particular programas que utilizan direccionamiento indexado, entonces tiene que encontrar por lo menos un par de lugares de memoria disponible en la página cero. Si mira a su alrededor con cuidado, podrá encontrar allí algunos lugares libres.



El sistema operativo de su Atari consume la mayor parte de la página cero. Hay un par de pequeños bloques de memoria que no son utilizados por el sistema operativo, pero no siempre están disponibles para los programas escritos por el usuario, ya que a menudo se destinan a otros usos. Por ejemplo, cuando se escribe un programa con un ensamblador, éste siempre utiliza algún lugar de la página cero. Si el programa está diseñado para ser llamado desde el BASIC, el intérprete que tendrá que usar consumirá un poco más de la página cero.

Las rutinas matemáticas de punto flotante del sistema operativo de su Atari también consumen un bloque de memoria en la página cero. Sin embargo, si se escriben programas que no utilizan el paquete de punto flotante, entonces el bloque de memoria que se reserva para el paquete quedará disponible. En concreto, estas son las posiciones de memoria en la página cero y las condiciones en que las puede utilizar:

Mapa de memoria de la Página Cero

- $00 - $AF - Reservado para uso del sistema operativo.
- $B0 - $CF - Bytes dejados libre por el cartucho Assembler Editor.
- $CB - $D1 - Bytes dejados libre por el cartucho BASIC.
- $D4 - $FF - libres si no usa su paquete de punto flotante del sistema operativo, el cartucho Atari Assembler Editor o un programa BASIC que utilice las rutinas de punto flotante del OS de Atari.

Posiciones que puede usar en la Página Cero

En los programas presentados en este libro, todas las posiciones de la página cero que se han utilizado han caído en el bloque de memoria que se extiende desde la dirección $B0 a la $CF. Fíjese en la tabla anterior, y verá por qué. Al escribir un programa utilizando el cartucho Atari Assembler Editor, el bloque que va desde la dirección $B0 a la $CF es la única parte de la página cero que no usa ya sea el sistema operativo de su computador o su cartucho Assembler Editor. (El ensamblador MAC/65 utiliza menos la página cero, pero los programas de este libro fueron escritos para ser compatibles con ambos ensambladores.) Si está escribiendo un programa diseñado para ser llamado desde el BASIC, la parte de la página cero que puede usar es aún más pequeña. Así, su espacio libre se extenderá sólo desde la dirección de memoria $CB hasta la dirección de memoria $D1. ¡Eso da un total de siete bytes de la página cero que se pueden utilizar en su programa! Hay dos maneras fáciles de eliminar esta limitación. Hay que escribir programas que utilizan muy poco la página cero, ¡o hay que escribir programas que no tienen que ser llamados desde el BASIC!

Las direcciones de memoria de la $100 a la $6FF
RAM del Sistema Operativo

Las direcciones de memoria de la $100 a la $5FF de su computador están reservadas para la memoria RAM del sistema operativo. Este bloque de memoria está dividido de la siguiente manera:

- De la dirección $100 a la $1FF - La pila de hardware de su computador. Puede usar este bloque de memoria, pero sólo para operaciones de manipulación de la pila. Acuérdese: PLA, PHA, PLP, PHP, JSR y RTS.
- De la dirección $200 a la $3BF - IOCB (bloques de control de entrada/salida) y otras variables del OS. El computador utiliza esta sección de memoria

principalmente para comunicarse con los dispositivos de entrada y de salida. No está disponible para ser usada por los programas escritos por el usuario.

- De la dirección $3C0 a la $3E7 - Buffer de impresión, donde se almacena la información cuando ésta se dirige a la impresora.

- De la dirección $3E8 al $3FC - Reservado para el sistema operativo, no se encuentra disponible para usted.

- De la dirección $3FD a la $47F - Buffer del casete, un área de almacenamiento para los datos que viajan entre su computador y la grabadora de casetes.

- De la dirección $480 a la $57D - Reservado para uso del cartucho BASIC. Puede ser utilizado por los programas en lenguaje ensamblador que no son llamados desde el BASIC.

- De la dirección $57E a la $5FF: - paquete de punto flotante del OS. Utilizado por el BASIC. También puede ser utilizado por programas escritos por usuario en lenguaje ensamblador. Está libre para otros usos en los programas en lenguaje ensamblador, si no se utilizan las rutinas de punto flotante y el cartucho BASIC.

- De la dirección $600 a la $6FF - "Página 6". Por lo general están disponibles para ser usadas por los programas en lenguaje ensamblador del usuario. Sin embargo, hay una excepción importante: Cuando se utiliza la instrucción INPUT en un programa BASIC, y se introducen más de 128 caracteres a través del teclado, los caracteres excedentes serán almacenados en la página 6. En este caso, el código objeto almacenado en las direcciones de la $600 a la $67F podría ser borrado. sin embargo, las direcciones de la $680 a la $6FF están a disposición de los programas escritos por el usuario de manera incondicional.

Direcciones de memoria de la $700 hasta "MEMLO"

La RAM dedicada del DOS

A partir de la dirección $700 se encuentra un bloque de memoria que está reservado para el uso del sistema operativo del disco de su computador. El tamaño de este bloque de memoria se ve afectado por una serie de factores, incluyendo cuántas unidades de disco se están utilizando, y si actualmente está utilizando los programas utilitarios del disco que aparecen en el menú del DOS de su computador. Debido a que el tamaño de este bloque de memoria varía tan ampliamente de computador en computador, y de aplicación en aplicación, su Atari ha sido equipado con una variable especial de 16 bits que puede decirle a simple vista cómo están relacionados el DOS y su bloque de memoria. Esta variable se llama MEMLO, y está almacenada en las direcciones de memoria de la $2E7 a la $2E8 (743 y 744 en notación decimal). El valor de 16-bits que tiene MEMLO es un número muy importante. No es sólo la dirección donde terminan las rutinas del DOS de su

computador, también es la dirección donde en su computador comienza el mayor bloque de memoria direccionable por el usuario. Una vez que sepa lo que significa el valor de MEMLO, sabrá exactamente dónde comenzar el código objeto de sus programas en lenguaje de máquina.

De "MEMLO" a "MEMTOP"
RAM Libre

La memoria RAM que puede utilizar libremente se extiende desde la variable llamada MEMLO (direcciones $2E7 y $2E8) hasta otra variable de 16-bits llamada, lógicamente, MEMTOP. Puede ver el valor que contiene MEMTOP echando una mirada al contenido de las direcciones de memoria $2E5 y $2E6 (741 y 742 en notación decimal). Una vez que sepa cuál es el valor de MEMTOP, sabrá el límite superior del bloque de memoria en el que puede almacenar sus programas en lenguaje ensamblador.

Por encima de "MEMTOP"
Memoria de la Pantalla

El bloque de memoria que se extiende desde MEMTOP hacia arriba corresponde al área de despliegue de su computador ("display"), un área reservada para los datos que éste utiliza para crear su visualización por la pantalla. Los programas que crean su propio display personalizado pueden sobrescribir este bloque de memoria RAM, pero si utiliza el display que viene incorporado en su computador, tendrá que permanecer fuera de esta sección de memoria, porque allí es donde se encuentra el display.

De la dirección $8000 a la $9FFF
Ranura de cartuchos B

Cuando se diseñó el computador Atari 800 (no el XL), este bloque de memoria se asignó al "Cartucho B", la ranura de la derecha del par de ranuras para los cartuchos. Sin embargo, la ranura de cartuchos B fue utilizada por sólo uno o dos programas escritos para el Atari. Así que los equipos más nuevos, los modelos 1200XL y posteriores, han sido diseñados con una sola ranura para cartuchos. Esto significa que las direcciones de memoria de la $8000 a la $9FFF, originalmente diseñadas para el "Cartucho B," se encuentran disponibles para ser usadas en programas escritos por el usuario.

De la $8000 a la $BFFF
Ranura de cartuchos A

La ranura de Cartuchos A es la ranura que usan la mayoría de los cartuchos de Atari, el cartucho Atari Assembler Editor, y todos los otros programas disponibles en cartucho. Muchos de los programas basados en disco también usan este bloque de memoria. Al escribir programas que usan cartuchos o programas utilitarios que ocupan este bloque de memoria, no hay una manera fácil en que pueda utilizar este espacio para sus programas.

Sin embargo, si es bueno escribiendo código reubicable, no hay razón por la que no pueda usar este bloque de memoria. ¡Después de todo, ya lo han hecho otros programadores de ensamblador de Atari! Normalmente esta área va desde la dirección $A000 a la $BFFF, pero algunas veces puede ir desde la dirección $8000 a la $BFFF.

De la dirección $C000 a la $CFFF

No se utiliza en el Atari 400 ni en el 800, ya que no tienen esta posición de memoria RAM. Esta área es usada por el sistema operativo en los nuevos modelos. Entre acá bajo su propio riesgo, no se recomienda su uso para programas escritos por el usuario.

De la dirección $D000 a la $D7FF
Registros de lectura/escritura del hardware Atari

No se encuentra disponible para ser usado por programas escritos por el usuario.

De la dirección $D800 a la $DFFF
ROM del Punto Flotante

Disponible para ser usada por los programas escritos por el usuario si es que no se utiliza el paquete de rutinas de punto flotante del sistema operativo, y si no se utilizan rutinas BASIC que requieran las rutinas de PF. Sólo está disponible en la línea XL; el Atari 400 y 800 no tienen memoria RAM disponible aquí.

De la dirección $E000 a la $FFFF
ROM del Sistema Operativo

No está disponible para ser usada por programas escritos por el usuario.

El problema de la asignación de la memoria

Una vez que conozca su mapa de memoria Atari como la palma de su mano, estará casi listo para comenzar a reservar la memoria para sus programas en lenguaje ensamblador. Casi, pero no del todo. Primero tendrá que aprender a resolver dos grandes problemas que pueden ser un verdadero dolor de cabeza para los programadores en lenguaje ensamblador del Atari. Estos dos problemas son:

- Asegurarse de que el código fuente y el código objeto de sus programas no se sobre sobrescriban mutuamente.

- Mantener separados en la memoria del computador los programas BASIC y los programas en lenguaje de máquina.

En realidad, estos dos problemas no son difíciles de resolver. Pero parecen ser más complicados de lo que son debido al confuso sistema desarrollado por Atari para seguirle la pista de la dirección más baja de la memoria libre.

En el sistema operativo de su computador existen dos variables, o punteros, que fueron diseñados para ayudarle a averiguar dónde pueden comenzar en la memoria sus programas en lenguaje de máquina. Una de estas variables se llama LOMEM, y la otra se llama MEMLO. Si piensa que esto es un poco confuso, verá que es sólo el principio. A veces LOMEM y MEMLO pueden ser intercambiados y, a veces no. Si bien sus abreviaturas son más que confusas, sus nombres completos son francamente engañosos. En las páginas D-1 y D-2 de su Atari Basic Reference Manual se identifica a MEMLO como el puntero a la memoria baja del sistema operativo de su computador, y LOMEM se identifica como el puntero a la memoria baja del BASIC de su Atari. A menos que quiera terminar totalmente desconcertado, no le haga caso a ninguno de estos nombres. He aquí cómo realmente los punteros LOMEM y MEMLO de su computador funcionan, y qué es lo que le pueden decir.

El puntero LOMEM

El puntero LOMEM de su Atari es una variable de 16 bits que se encuentra ubicada en las direcciones de memoria $80 y $81 (o 128 y 129 en notación decimal). LOMEM siempre contiene la dirección de comienzo de un bloque de memoria de su computador llamado el "buffer de edición de texto". El buffer de edición de texto es un buffer especial diseñado para contener el texto ATASCII que está siendo escrito y editado. Al escribir o editar un programa BASIC, éste se almacena en el buffer de edición de texto hasta que esté listo para ser ejecutado. El buffer de edición de texto también se utiliza para almacenar el código fuente de sus programas en lenguaje ensamblador.

Al encender su computador, la dirección almacenada en LOMEM corresponde a la dirección más baja de la memoria RAM. Es decir, la dirección más baja (sin incluir la página 6) en la que sus programas pueden comenzar de forma segura. Si no tiene ninguna unidad de disco conectada al computador, entonces el valor de LOMEM será $0700.

Si tiene una o más unidades de disco conectadas a su computador, y están encendidas, entonces el sistema operativo del disco (DOS) se almacenará en el bloque de memoria RAM que se encuentra justo debajo de LOMEM.

Cambiar el puntero LOMEM

A pesar de que el valor de LOMEM se ajusta automáticamente a un valor predeterminado cada vez que enciende su computador, puede cambiar su valor cuando lo desee. Cuando el valor de LOMEM cambia, la dirección inicial del buffer de edición de texto de su computador cambia automáticamente a la dirección que se ha cargado en el puntero

LOMEM. Esto significa que puede cambiar la ubicación del buffer de edición de texto de su computador cuando lo desee, solamente cargado una nueva dirección de 16 bits en el puntero LOMEM.

¿Por qué querría usted cambiar la ubicación del buffer de edición de texto? La razón más común es para mantener el código fuente y objeto de los programas separados el uno del otro mientras el código fuente está siendo escrito, editado y ensamblado. Para entender cómo se puede usar LOMEM para mantener el código fuente y objeto separados, se debe comprender cómo se relacionan LOMEM y MEMLO.

El puntero MEMLO

MEMLO también es un valor de 16-bits, pero se almacena en las direcciones de memoria $2E7 y $2E8 de su computador (ó 743 y 744 en decimal). Cuando enciende su computador, sin haber introducido anteriormente un cartucho o un disco, el puntero MEMLO siempre contiene la dirección más baja de la memoria RAM libre, o sea, la dirección más baja en la que pueden empezar los programas escritos por el usuario. Esto significa que cuando enciende su computador, MEMLO y LOMEM contienen exactamente la misma dirección: la dirección más baja de la memoria RAM libre. Esta dirección es también la dirección de comienzo del buffer de edición de texto de su computador.

Ahora supongamos que está sentado frente a su computador, y que su equipo, su ensamblador, las unidades de disco y su disco de datos están todos listos y funcionando. Ahora supongamos que el mensaje EDIT de su ensamblador acaba de salir por pantalla, y que está listo para comenzar a escribir algún código fuente. Dado que acaba de encender su computador, los punteros LOMEM y MEMLO contendrán la misma dirección cuando comience su sesión de edición: la dirección libre más baja de la memoria RAM de su computador. Puesto que no ha cambiado ningún valor por defecto, la dirección también será la dirección inicial del buffer de edición de texto de su equipo.

Por lo tanto cuando comience a escribir el código fuente, este siempre comenzará en la dirección libre más baja disponible en la memoria de su computador, lo que nos lleva a una conclusión lamentable, pero evidente: Cuando se escribe un programa en lenguaje ensamblador con el ensamblador MAC/65 o el cartucho Atari Assembler Editor, no puede comenzar el código objeto de su programa en la dirección que figura en los punteros LOMEM y MEMLO: si trata de hacerlo, sus códigos fuente y objeto intentarán sobrescribirse mutuamente, ¡Y su equipo le recompensará con un mensaje de error!

Solucionando el problema

Entonces, ¿qué puede hacer el pobre programador? Bueno, puede hacer un par de cosas. Por ejemplo, puede utilizar un comando especial llamado SIZE ("tamaño") que viene como un bono especial tanto en el ensamblador MAC/65 como en el cartucho Atari Assembler Editor. Es fácil utilizar el comando SIZE. Todo lo que tiene que hacer es poner

el ensamblador en modo Edición, escriba la palabra SIZE, y presione la tecla RETURN. El computador imprimirá una línea en la pantalla que se ve más o menos así:

```
1CFC 2062 9C1F
```

El significado de la línea "SIZE"

El primero de estos tres números, 1CFC, es el valor actual del puntero LOMEM de su computador - la dirección utilizable más baja de la memoria RAM de su computador. El segundo número, 2062 en nuestro ejemplo, es el valor del puntero MEMLO de su computador - la dirección en la que actualmente termina el buffer de edición de texto. (Se utilizó la palabra "actualmente" debido a que la longitud del buffer de edición de texto de su Atari puede variar: a medida que escribe su código fuente, el buffer de edición de texto crece. Cuando se elimina el código fuente, se contrae).

El tercer número de la línea SIZE de su ensamblador (9C1F en nuestra línea de ejemplo) es el valor de otro puntero importante -- MEMTOP, la dirección de memoria más alta que puede ser utilizada con seguridad en un programa escrito por el usuario. (Justo encima de MEMTOP es donde comienza la memoria de visualización ("display") de la pantalla de su computador.)

Tres hechos básicos

El comando SIZE de su ensamblador, entonces, le proporciona tres hechos importantes que pueden ayudarle con sus problemas de asignación de memoria. Le dice:

- Dónde empieza el código fuente de su programa.
- Dónde termina el código fuente de su programa.
- Cuánta RAM libre hay entre el final del código fuente de su programa y el inicio de la memoria de visualización ("display") de la pantalla de su computador.

Una advertencia

Sin embargo, si utiliza el comando SIZE para decidir dónde almacenar su código objeto, tenemos otra advertencia: La mayoría de programas en lenguaje ensamblador producen una tabla de símbolos: una lista de etiquetas que se utilizan dentro del programa y sus correspondientes direcciones de memoria. Cuando se escribe un programa que contiene etiquetas usando el Atari Assembler Editor, el ensamblador almacenará automáticamente la tabla de símbolos justo encima del buffer de edición de texto de su computador. Así que cuando utilice el Atari Assembler Editor para escribir un programa que produce una tabla de símbolos, siempre hay que dejar algún espacio entre el final del buffer de edición de texto (el segundo número de la línea SIZE) y el comienzo del código objeto de su

programa. De lo contrario su tabla de símbolos y el código objeto de su programa pueden sobrescribirse mutuamente, con resultados potencialmente desastrosos.

No hay necesidad de adivinar

Afortunadamente, no tiene que adivinar cuánto espacio se necesita para almacenar una tabla de símbolos: usted mismo puede calcularlo. Sólo necesita tres bytes para cada etiqueta que se usa en su programa, más un byte por cada carácter digitado en cada etiqueta. Esto suena a que se debe hacer una gran cantidad de cálculos, y así es. Pero si lo hace bastante tiempo, a la larga se volverá muy eficiente calculando la longitud de las tablas de símbolos.

¡Hay una manera más fácil!

Ahora que sabe todo esto, he aquí una buena noticia. Hay otro comando que puede usarse tanto en el MAC/65 y en el Atari Assembler Editor, un comando llamado LOMEM, que puede hacer que todo este asunto de la asignación de memoria sea mucho más fácil. He aquí cómo utilizar el comando LOMEM: Cuando haya cargado su ensamblador en la memoria RAM (o cuando haya introducido el cartucho de su ensamblador y haya encendido su computador), escriba la palabra LOMEM seguido de un número hexadecimal de la siguiente manera:

```
LOMEM $5000
```

Luego presione su tecla RETURN. Este sencillo procedimiento restablecerá automáticamente el puntero LOMEM de su computador, y pondrá el código fuente que posteriormente escriba por sobre el código objeto que se genera, en lugar de debajo de éste. A continuación, puede almacenar su código fuente de cualquier lugar que desee, en los espacios abiertos encima de su código de máquina, en vez de debajo de éste, donde casi no hay espacio.

Con el comando LOMEM, nunca tendrá que preocuparse de cuál es el valor actual de MEMLO, y nunca tendrá que contar el número de caracteres escritos en una tabla de símbolos. Sin embargo, hay que recordar que si desea utilizar el comando LOMEM, debe utilizarlo antes de empezar a escribir cualquier programa. Si utiliza el comando LOMEM con el ensamblador MAC/65, se borrará todo lo almacenado en la memoria RAM, de manera similar al comando NEW. Cuando se está escribiendo un programa con el cartucho Atari Assembler Editor, LOMEM debe ser el primer comando que se use apenas haya encendido su computador. De lo contrario, simplemente no funcionará. Si se olvida de hacer esto y aun así desea utilizar el comando LOMEM, tendrá que apagar su equipo y volver a encenderlo nuevamente.

Otro problema de gestión de memoria

También puede encontrar problemas de asignación de memoria cuando se mezcla el lenguaje ensamblador con el BASIC, o sea, cuando se escribe un programa en lenguaje ensamblador que está diseñado para ser llamado desde un programa BASIC. Cuando desee llamar a un programa en lenguaje de máquina desde el BASIC, obviamente es necesario que ambos programas estén presentes al mismo tiempo en su computador. Por desgracia también es evidente que los dos programas no pueden comenzar en la misma dirección. Si lo hicieran, un programa sobrescribiría al otro, y darían como resultado un caos absoluto. Afortunadamente, hay maneras de resolver este problema.

Cambiando el puntero MEMLO

Cuando se carga un programa BASIC en la memoria, su computador utiliza el valor de MEMLO para determinar dónde se debe almacenar el programa. Si MEMLO apunta a la dirección libre más baja de la RAM en su computador cuando se cargue un programa BASIC, el programa será cargado en la memoria de su computador comenzando en esa dirección. Pero si MEMLO apunta a una dirección más alta cuando se carga en memoria un programa BASIC, entonces el programa se cargará en la memoria RAM comenzando en esa dirección.

Entonces, obviamente, la manera de hacer que un programa BASIC no sobre escriba un programa en lenguaje de máquina almacenado en la memoria baja es cambiando MEMLO para que apunte a una dirección mayor antes de que se cargue el programa BASIC. Es fácil cambiar MEMLO para que apunte a un valor más alto antes de que se cargue un programa BASIC. Todo lo que tiene que hacer es usar una rutina como esta:

CAMBIANDO EL VALOR DEL PUNTERO MEMLO

```
10   ;
20   ;NEWMEMLO.SRC
30   ;
40      *=$0600
50   ;
60   NEWMLO=$5000 ;NUEVA DIRECCIÓN DE MEMLO
65   MEMLO=$2E7   ;DIRECCIÓN DEL PUNTERO DE MEMLO
70   ;
80      LDA #NEWMLO
90      STA MEMLO
100     LDA #NEWMLO/256
110     STA MEMLO+1
120     RTS
```

¿Qué es lo que acaba de hacer?

Cuando ensamble y ejecute esta rutina, se almacenará una nueva dirección en el puntero MEMLO de su computador, en este caso, $5000. Si a continuación carga un programa BASIC en la memoria de su computador, la dirección inicial de dicho programa no será la dirección más baja en la memoria RAM libre, como sería normalmente. En cambio, el programa BASIC se iniciará en la dirección de memoria $5000. Esto reservará un gran bloque de memoria para los programas en lenguaje de máquina escritos por el usuario: el bloque se extiende desde los bytes más bajos de la memoria RAM libre (el valor de LOMEM) hasta la dirección de memoria $ 4FFF.

Una manera mejor

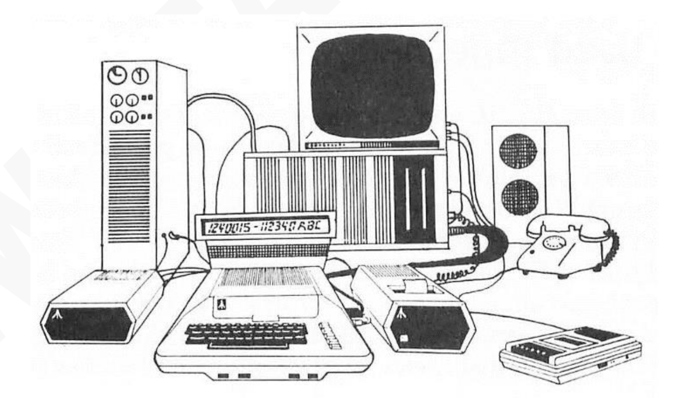
A pesar de que la rutina que hemos presentado sólo funcionará bien en los programas que ejecute usted mismo, puede no ser adecuada para programas diseñados para ser ejecutados por otras personas. Eso es porque el puntero MEMLO se establece no sólo cuando se enciende el computador sino también cuando se oprime el botón RESET. Así que si el botón de RESET es presionado accidentalmente por el usuario del programa, MEMLO se restablecerá a su valor predeterminado. Un programa más complejo para configurar MEMLO y que es inmune a los accidentes, como el presionar accidentalmente el botón RESET, se encuentra en las páginas de la 8 a la 11 del De Re Atari, la guía del programador en lenguaje ensamblador publicada por Atari. Este programa, que puede digitar, incluso se puede ejecutar como una rutina AUTORUN.SYS, y es casi tan transparente al usuario como debería ser este tipo de programa.

Capítulo Doce
Las Entradas y Salidas (E/S) y Usted

Tipos de dispositivos de E/S

A su computador Atari se le pueden conectar muchos tipos de dispositivos de E/S. Sin embargo, hay siete tipos específicos de dispositivos con los que puede comunicarse, mediante procedimientos y comandos específicos, tanto desde el BASIC como del lenguaje ensamblador. Cada uno de estos siete tipos de dispositivos tiene una abreviación única de una letra, o nombre de dispositivo, por el cual pueden ser direccionados tanto por el BASIC como por el lenguaje ensamblador. Estos siete tipos de dispositivos, y sus correspondientes nombres, son:

- Teclado (K:)
- Impresora (P:)
- Grabadora de casetes (C:)
- Unidades de disco (D:) (o si se utiliza más de una unidad de disco, D1:, D2:, D3: y D4:)
- Editor de Pantalla (E:)
- Monitor de TV (Pantalla) (S:)
- Interfaz serie RS-232 (R:)



Observe los dos puntos que vienen a continuación de la letra en cada una de estas abreviaturas. Los dos puntos son una parte integral de cada nombre de dispositivo, y no pueden omitirse.

Las Ocho Operaciones de E/S de su Atari

Tanto en el BASIC como en el ensamblador tenemos ocho operaciones de E/S que se pueden realizar usando las siete abreviaturas, o nombres de dispositivo, mencionados anteriormente. Estas ocho operaciones de E/S son las siguientes:

- OPEN (Abrir un dispositivo determinado).
- CLOSE (Cerrar un dispositivo determinado).
- GET CHARACTER (Leer un carácter desde un dispositivo o archivo especificado).
- PUT CHARACTER (Escribir un carácter a un dispositivo o archivo especificado).
- READ RECORD (Leer el siguiente registro (una cadena que debe terminar con un carácter de retorno [$9B]) desde un dispositivo o archivo especificado).
- WRITE RECORD (Escribir un registro (una cadena que debe terminar con un carácter de retorno [$9B]) en un dispositivo o archivo especificado).
- ESTADO (Obtener el estado de un dispositivo determinado).
- SPECIAL (Para realizar una operación especial sobre el dispositivo especificado. Se usa principalmente en la gestión de archivos y operaciones sobre la interfaz serie RS-232).

Como se usan los nombres y las operaciones de E/S

Tanto en el BASIC como en el ensamblador, todas las operaciones de E/S mencionadas anteriormente están diseñadas para ser utilizadas por medio de un sistema centralizado de interfaz de periféricos llamado "Central I/O utility - Utilitario de E/S Central", o CIO. El sistema CIO de Atari, como la mayoría de los sistemas de interfaz de periféricos, fue diseñado para manejar secuencias de bytes de datos llamados archivos. Un archivo puede contener datos, texto, o ambos, y puede o no puede estar organizado por registros, cadenas de texto o datos separados por caracteres de final de línea (código ATASCII $9B). Algunos archivos, como los archivos grabados en disco, pueden tener nombres individuales (como por ejemplo "D1: TESTIT.SRC"). Otros archivos, como los utilizados con el editor de pantalla de su Atari o la impresora, no tienen nombres propios, pero uno puede comunicarse con ellos simplemente usando el nombre del dispositivo en el que aparecen, por ejemplo, "E:" o "P:".

Tanto el BASIC como el ensamblador permiten a los programadores acceder hasta ocho dispositivos y/o archivos distintos al mismo tiempo. Este acceso se ofrece a través de ocho bloques de memoria que se llaman "Input/Output Control Blocks - Bloques de Control de Entrada/Salida", o IOCBs. En el ensamblador, al igual que en el BASIC, los ocho IOCBs están numerados del 0 al 7. En ambos lenguajes, cualquier número IOCB libre puede ser asignado a cualquier dispositivo de E/S, aunque cuando su computador Atari se enciende por primera vez el IOCB #0 siempre es asignado al editor de la pantalla. El IOCB #0 también corresponde al número de IOCB predeterminado del editor de pantalla.

La apertura de un dispositivo

Tanto en el BASIC como en el ensamblador, a los dispositivos de E/S se les asignan números IOCB cuando se comunica con ellos por primera vez, o sea, al momento de ser abiertos. Cuando un dispositivo es abierto por primera vez, ya sea para operaciones de lectura o escritura, se le debe asignar un número IOCB. Una vez que se le ha asignado un número IOCB a un dispositivo, el dispositivo puede ser referenciado por este número hasta que se utilice un comando para cerrar el dispositivo. Una vez cerrado el dispositivo, el número de IOCB que le fue asignado se libera de nuevo, y puede ser utilizado para abrir otro dispositivo en su computador.

El lenguaje ensamblador carece de comandos IOCB

En el BASIC de Atari se proporcionan comandos específicos para abrir, cerrar, leer y escribir en cualquier dispositivo de E/S que pueda conectarse a su computador. No existen comandos de esta índole en el lenguaje ensamblador del 6502. El sistema IOCB utilizado en los computadores Atari provee al programador en lenguaje ensamblador un medio para manejar todos los dispositivos de E/S que puedan conectarse a su computador Atari. Se pueden manejar de una manera que sea relativamente fácil de administrar y de entender.

La apertura de un dispositivo mediante el Atari BASIC

No es difícil abrir un dispositivo o un archivo en el BASIC de Atari. Para abrir un dispositivo o un archivo, todo lo que tiene que hacer es escribir una línea utilizando la siguiente fórmula.

```
10 OPEN #n,nl,n2,filespec
```

El siguiente es un ejemplo de una instrucción BASIC de Atari que ha sido escrita utilizando la fórmula IOCB estándar.

```
10 OPEN #2,8,0,"Dl:TESTIT.BAS"
```

Como puede ver, hay cinco componentes en una instrucción OPEN en el BASIC de Atari: El comando OPEN en sí, una serie de tres parámetros separados por comas, y un nombre de dispositivo, más un nombre de archivo, si éste es necesario. El símbolo "#" debe aparecer obligatoriamente antes del primer parámetro después de la instrucción OPEN. Además, dos puntos deben ir obligatoriamente después del nombre del dispositivo. El nombre del dispositivo y el nombre del archivo, si son necesarios, van obligatoriamente entre comillas. Los significados de los cinco componentes de una instrucción OPEN se explican a continuación.

1. "OPEN" - el comando OPEN.

2. "#n" (#2 en nuestro ejemplo) - El número IOCB. Este número, como ya hemos señalado, va desde el 0 al 7. "#2" en este caso significa "IOCB #2".

3. "n1" (8 en nuestro ejemplo) - Un número de código para un tipo específico de operación de entrada o de salida. En nuestra orden OPEN de ejemplo, el "8" en esta posición corresponde al número de código para una operación de salida (abrir para escritura).

4. "n2" (0 en nuestro ejemplo) - Un código auxiliar dependiente del dispositivo que a veces se utiliza para diversos fines (Aunque en este caso no se utiliza).

5. "filespec" - Un nombre del dispositivo seguido de un nombre de archivo, si es que es necesario. En nuestro ejemplo, "D1:TESTIT.BAS" se refiere a un archivo llamado TESTIT.BAS que nuestro equipo espera encontrar almacenado en el diskette que se encuentra en la unidad de disco 1.

Cómo BASIC procesa el comando "OPEN"

Cuando al procesar un programa BASIC el computador se encuentra con un comando OPEN, lleva a cabo una serie de operaciones estándares que usan los valores de cada uno de los cuatro parámetros de la instrucción OPEN. Cuando se hayan completado todas estas operaciones, BASIC ejecuta una subrutina especial del sistema operativo llamado "Vector CIO", o CIOV. La subrutina CIOV abre automáticamente el dispositivo en cuestión, haciendo referencia a los parámetros que figuran en la instrucción OPEN (y que ahora están almacenados en ubicaciones de memoria determinadas) a fin de asegurarse de que se abra el dispositivo correcto para el tipo de acceso indicado en la instrucción OPEN.

Ventajas de las operaciones de E/S en Lenguaje Ensamblador

Para entender cómo se abre un dispositivo utilizando el lenguaje ensamblador, es útil saber cómo se abren desde el BASIC de Atari. Esto porque tanto los programas en BASIC como en lenguaje ensamblador abren los dispositivos exactamente de la misma manera. La única diferencia es que cuando se abre un dispositivo desde el BASIC, el intérprete BASIC hace la mayoría del trabajo por usted. Cuando se utiliza el lenguaje ensamblador, tiene que hacer todo el trabajo usted mismo. Afortunadamente, hay una recompensa por hacer todo este trabajo extra. Cuando se controla el sistema CIO utilizando el lenguaje ensamblador, se tiene mucho más control sobre el sistema que cuando se le permite al BASIC hacer todo el trabajo.

Abrir un dispositivo utilizando Lenguaje Ensamblador

Ahora echemos un vistazo a cómo exactamente se abren, leen, escriben y cierran los dispositivos tanto desde el BASIC como desde el lenguaje ensamblador.

Otra mirada a los IOCBs

Como hemos señalado, las operaciones de E/S de su computador Atari son controladas mediante una serie de ocho bloques de control de E/S, o IOCBs. Cada uno de estos bloques de control de E/S es un bloque de memoria real en su computador. Cada IOCB tiene 16 bytes de largo, y cada byte en cada IOCB tiene un nombre y una función específica. Por otra parte, cada byte en cada IOCB tiene el mismo nombre, y realiza el mismo tipo de función que el byte correspondiente en cada otro IOCB. Esto es importante, así que vamos a decirlo de nuevo de una manera distinta: Cada byte en cada IOCB de su computador tiene el mismo nombre, y realiza el mismo tipo de función que el byte con la misma ubicación (offset) en cada otro IOCB.

Direccionamiento Indirecto en las Operaciones con IOCB

La razón de que esto sea importante es porque el direccionamiento indirecto se utiliza a menudo en las operaciones de IOCB. El Direccionamiento Indirecto es una técnica en la que se solicita una posición de memoria usando un valor de desplazamiento (offset) almacenado en el registro X o Y del procesador 6502. El que los desplazamientos de todos los bytes en todos los IOCBs de su Atari se corresponden entre sí hace que el modo de direccionamiento indirecto sea muy fácil de utilizar en las operaciones de IOCB de su Atari.

Los 16 bytes de un IOCB

Este concepto es mucho más fácil de entender cuando se dan algunos ejemplos. Así que utilizaremos un programa de lenguaje ensamblador real para explicar el sistema de E/S de su Atari. Si este programa le parece familiar, es porque es casi exactamente igual al que le pedí que escribiera en el capítulo 7. Es el mismo que utilizamos para imprimir un mensaje en la pantalla. Si todavía tiene ese programa almacenado en un disco, puede cargarlo en su computador, y con sólo unos pocos cambios, puede convertirlo en una réplica exacta del programa que viene a continuación. De esta manera no tendrá que escribirlo de nuevo.

PROGRAMA PARA IMPRIMIR EN LA PANTALLA

```
10   ;
20   .TITLE "RUTINA PRNTSC"
30   .PAGE "RUTINA PARA IMPRIMIR EN LA PANTALLA"
40   ;
```

```
50      *=$5000
60   ;
70   BUFLEN=255 ;(SE EXPANDE MAS ALLA DE LOS LIMITES ANTERIORES)
80   ;
90   EOL=$9B ;CODIGO ATASCII DEL CARACTER FIN DE LINEA
100  ;
110  OPEN=$03 ;CODIGO PARA ABRIR UN DISPOSITIVO O ARCHIVO
120  OWRIT=$0B ;CODIGO PARA "OPERACIONES DE LECTURA"
130  PUTCHR=$0B ;CODIGO PARA "ESCRIBIR CARACTER"
140  CLOSE=$0C ;CODIGO PARA CERRAR UN DISPOSITIVO
150  ;
160  IOCB2=$20 ;DESPLAZAMIENTO PARA EL IOCB NO. 2
170  ICCOM=$342 ;BYTE DE COMANDO (CONTROLA LAS OPERACIONES CIO)
180  ICBAL=$344 ;DIRECCION DEL BUFFER (BYTE BAJO)
190  ICBAH=$345 ;DIRECCION DEL BUFFER(BYTE ALTO)
200  ICBLL=$348 ;LARGO DE BUFFER (BYTE BAJO)
210  ICBLH=$349 ;LARGO DE BUFFER (BYTE ALTO)
220  ICAX1=$34A ;BYTE AUXILIAR NO.1
230  ICAX2=$34B ;BYTE AUXILIAR NO.2
235  ;
240  CIOV=$E456 ;VECTOR CIO
250  ;
260  DEVNAM .BYTE "E:",EOL
270  ;
280  OSCR ;RUTINA PARA ABRIR LA PANTALLA
290      LDX #IOCB2
300      LDA #OPEN
310      STA ICCOM,X
320  ;
330      LDA #DEVNAM&255
340      STA ICBAL,X
350      LDA #DEVNAM/256
360      STA ICBAH,X
370  ;
380      LDA #OWRIT
390      STA ICAX1,X
400      LDA #0
410      STA ICAX2,X
420      JSR CIOV
430  ;
440      LDA #PUTCHR
450      STA ICCOM,X
460  ;
470      LDA #TXTBUF&255
480      STA ICBAL,X
490      LDA #TXTBUF/256
500      STA ICBAH,X
510      RTS
520  ;
530  PRNT
540      LDX #IOCB2
550      LDA #BUFLEN&255
```

```
560       STA ICBLL,X
570       LDA #BUFLEN/256
580       STA ICBLH,X
590       JSR CIOV
600       RTS
605  ;
610  CLOSED
620       LDX #IOCB2
630       LDA #CLOSE
640       STA ICCOM,X
650       JSR CIOV
660       RTS
670  ;
680  TXTBUF=*
690  ;
700       *=*+BUFLEN
710  ;
720       .END
```

"PRNTSC.SRC", línea por línea

Ahora vamos a mirar bien de cerca este programa y veremos, línea por línea cómo funciona. Empezaremos por las primeras tres líneas del programa, de la 290 hasta 310.

Inicialización de un dispositivo para ser usado por "OPEN"

```
290    LDX #IOCB2
300    LDA #OPEN
310    STA ICCOM,X
```

Sustituya los números literales de las variables en estas tres líneas, y así es como se verán.

```
290    LDX #$20
300    LDA #$03
310    STA $342,X
```

Estas tres instrucciones son todo lo que se necesita para abrir un dispositivo en lenguaje ensamblador de Atari. Para comprender lo que hacen, tiene que saber algo acerca de la estructura de un IOCB de su Atari. Como hemos señalado, hay ocho IOCBs en el sistema operativo de su Atari, y cada uno de ellos contiene 16 bytes (o $10 bytes en notación hexadecimal). Esto significa que para acceder al IOCB # 1, tiene que sumar 16 (o $ 10) bytes a la dirección de IOCB #0 y para acceder al IOCB #2, tiene que agregar 32 (o $ 20) bytes a la dirección del IOCB #0. En otras palabras, cuando se utiliza la dirección IOCB #0 como un punto de referencia (como lo hace el sistema CIO de Atari), el desplazamiento que tiene que utilizar es de 32 en decimal, o $ 20 en sistema hexadecimal. Aquí están todos los desplazamientos de IOCB utilizados en el sistema CIO de Atari:

Los ocho desplazamientos (offset) de los IOCB de su Atari

```
IOCB0=$00        IOCB4=$40
IOCB1=$10        IOCB5=$50
IOCB2=$20        IOCB6=$60
IOCB3=$30        IOCB7=$70
```

Ahora echemos otro vistazo a nuestra versión de valores literales de las tres primeras líneas de nuestro programa PRNTSC.SRC:

```
290    LDX #$20
300    LDA #$03
310    STA $342,X
```

Ahora puede comenzar a entender por qué se ha cargado el número $20 en el registro X de la línea 300. Obviamente, va a ser utilizado como un desplazamiento (offset) en la línea 320. Pero antes de pasar a la línea 320, echemos un vistazo a la línea 310, la línea que está justo en medio. En la línea 310 se carga el acumulador con el número $03, que ha sido identificado en la línea 110 del programa como el "código (token) para la apertura de un dispositivo". Ahora, ¿qué significa eso?

Códigos (tokens) de E/S

Bueno, en el sistema CIO de Atari, cada uno de las ocho operaciones de E/S descritas al principio de este capítulo puede ser identificada por un código o símbolo (token) de un dígito (hexadecimal). Aquí está la lista completa de códigos, y las operaciones a las que se refieren.

Token	Nombre	Función
$03	OPEN	Abre un dispositivo o archivo especificado.
$04	OREAD	Abre un dispositivo o archivo para operaciones de lectura.
$08	OWRITE	Abre un dispositivo o archivo para operaciones de escritura.
$05	GETREC	Lee un registro desde un dispositivo o archivo especificado.
$07	GETCHR	Lee un carácter desde un dispositivo o archivo especificado.
$09	PUTREC	Escribe un registro en un dispositivo o archivo especificado.
$OB	PUTCHR	Escribe un carácter en un dispositivo o archivo especificado.
$OC	CLOSE	Cierra un dispositivo o archivo especificado.

Explicación de la línea 310

Ahora puede ver lo que sucede en la línea 310 del programa PRNTSC.ASM. El acumulador se carga con el número $03, el símbolo correspondiente a "OPEN". En la línea 320, el símbolo OPEN se almacena en la dirección indirecta ICCOM,X (o $342, X). ¿A que corresponde exactamente esta dirección?

ICCOM es el nombre de uno de los 16 bytes de un IOCB. En concreto, ICCOM es el primer byte (el byte con desplazamiento (offset) cero) de cada IOCB. Observe la línea 170 del programa PRNTSC.ASM y verá que ICCOM está situado en la dirección de memoria $342, y se identifica como el "byte de instrucción (command byte)" en el sistema CIO de Atari. Se le llama byte de instrucción porque es el byte que debe ser referenciado cuando los dispositivos se inicializan, abren o cierran. ICCOM es el byte que apunta a un conjunto de subrutinas en el sistema operativo de su computador que realiza todas estas funciones.

Las direcciones de los IOCB

Dado que hasta el momento hemos nombrado todos los dispositivos Atari de E/S, los comandos de E/S, los desplazamientos (offset) de E/S y los códigos de operación de E/S, podríamos hacer una lista con ICCOM y el resto de los 16 bytes en cada uno de los IOCBS de su computador. Aquí está la lista completa de los bytes de cada IOCB.

Byte	Dirección	Nombre	Función
ICHID	$0340	Identificador del controlador	Predeterminado por el Sistema Operativo
ICDNO	$0341	Número de dispositivo	Predeterminado por el Sistema Operativo
ICCOM	$0342	Byte de Instrucción	Controla las operaciones CIO
ICSTK	$0343	Byte de Estado	Devuelve el estado de las operaciones
ICBAL	$0344	Buffer de Dirección, Byte Bajo	Tiene la dirección del buffer de texto
ICBAH	$0345	Buffer de Dirección, Byte Alto	Tiene la dirección del buffer de texto
ICFITL	$0346	Puntero sin uso	No se usa en los programas
ICPTH	$0347	Puntero sin uso	No se usa en los programas
ICBLL	$0348	Largo del Búfer, Byte Bajo	Tiene la longitud del buffer de texto
ICBLH	$0349	Largo del Búfer, Byte Alto	Tiene la longitud del buffer de texto
ICAX1	$034A	Byte auxiliar Nº 1	Selecciona operación de lectura o de escritura
ICAX2	$034B	Byte auxiliar Nº 2	Se utiliza para diversos fines
ICAX3	$034C	Byte auxiliar Nº 3	Usado solamente por el Sistema Operativo
ICAX4	$034D	Byte auxiliar Nº 4	Usado solamente por el Sistema Operativo
ICAX5	$034E	Byte auxiliar Nº 5	Usado solamente por el Sistema Operativo
ICAX6	$034F	Byte auxiliar Nº 6	Usado solamente por el Sistema Operativo

Entonces ahora puede entender las operaciones que se realizan en las líneas de la 300 a la 320 del programa PRNTSC.SRC.

```
290    LDX #IOCB2
300    LDA #OPEN
310    STA ICCOM,X
```

En la línea 290 se carga el registro X con el desplazamiento correspondiente al IOCB# 2: el número $20. En la línea 300 se carga el acumulador con el símbolo de la operación OPEN: el número $03. En la línea 310, el símbolo de la operación OPEN (el número $03) se almacena en ICCOM, X: el byte de instrucción del IOCB # 2. Después de unas pocas operaciones más, vamos a hacer una orden "JSR CIOV" (saltar a la subrutina), para que nuestro computador Atari salte al vector CIO y abra el IOCB #2, tal como le hemos indicado. Pero primero vamos a tener que establecer otros pocos parámetros para que nuestro equipo sepa exactamente qué tipo de operaciones realizar al abrir el IOCB #2. Así que ahora vamos a ver el resto de esta operación "OPEN".

Completando la Operación "OPEN"

```
330     LDA #DEVNAM&255
340     STA ICBAL,X
350     LDA #DEVNAM/256
360     STA ICBAH,X
370  ;
380     LDA #OWRIT
390     STA ICAX1,X
400     LDA #0
410     STA ICAX2,X
420     JSR CIOV
```

En las líneas de la 330 a la 360 se carga el buffer de texto del IOCB #2 con la dirección de la variable DEVNAM definida en la línea 260. La variable DEVNAM, como se puede ver mirando la línea 260, contiene los códigos ATASCII correspondientes a la cadena de caracteres "E:", el nombre del dispositivo del editor de pantalla de su Atari. Podríamos haber abierto el IOCB #2 para cualquier otro dispositivo de E/S de la misma manera. Por ejemplo, si quisiéramos utilizar el IOCB #2 como el IOCB de la impresora, tendríamos que escribir la línea 260 de la siguiente manera:

```
260 DEVNAM.BYTE "P:",EOL
```

Luego, en las líneas de la 330 a 360, la dirección de la cadena de caracteres ATASCII "P:", EOL se carga en ICBAL, X. Con este pequeño cambio en el programa PRNTSC, ¡Abrirá su impresora como un dispositivo de salida en vez de la pantalla! También puede utilizar este mismo procedimiento de programación para abrir un archivo específico en un disco de modo que usted pueda leer o escribir en él, ya sea un carácter a la vez, o un

registro a la vez. En el programa PRNTSC podríamos abrir un archivo de disco en lugar del editor de pantalla cambiando la línea 260 de la siguiente manera:

```
260 DEVNAM.BYTE "D1:TESTIT.BAS",EOL
```

Entonces, en lugar de abrir el editor de pantalla, nuestro programa abrirá el archivo TESTIT.BAS del disco (por supuesto, siempre y cuando exista una unidad de disco conectada a nuestro computador y que existan todas las otras condiciones necesarias para abrir tal archivo). Acabamos de ver dos ejemplos del tremendo poder del sistema CIO de Atari. Si bien el sistema puede parecer complejo a primera vista, su increíble versatilidad es un testimonio real de los conocimientos de programación que tienen los diseñadores de computadores Atari.

Continuemos

Sigamos ahora con nuestra operación "OPEN". En las líneas 390 y 400 se carga el número $08, el símbolo para "abrir un dispositivo para una operación de escritura" en el Byte Auxiliar Nº1 del IOCB #2. Podemos hacer que nuestro programa haga algo completamente diferente si se almacena el valor $04, el símbolo de "abrir para operaciones de lectura", en ICAXL, X en lugar del valor $08, el símbolo para "abrir para operaciones de escritura". Esa es otra demostración de la versatilidad del sistema CIO de Atari.

Hemos leído las líneas 410 y 420, en las que se limpia el Byte Auxiliar Nº2 del IOCB #2 (un byte que no se utiliza en esta rutina) asignándole un cero. Por último, en la línea 430, saltamos al vector CIO de su Atari que se encuentra en la dirección de memoria $E456. Con esta operación, hemos abierto el IOCB #2 para una operación de escritura en la pantalla del editor de su Atari. En otras palabras, hemos abierto el IOCB #2 para imprimir en la pantalla.

Imprimir un carácter

Sin embargo, todavía no hemos logrado imprimir un carácter en la pantalla. Para ello debemos llevar a cabo dos secuencias más de operaciones de E/S. Ahora que entendemos cómo funciona el sistema CIO de Atari, será fácil. Mire las líneas de la 450 a la 610 del programa PRNTSC.ASM.

```
430 ;
440    LDA #PUTCHR
450    STA ICCOM,X
460 ;
470    LDA #TXTBUF&255
480    STA ICBAL,X
490    LDA #TXTBUF/256
500    STA ICBAH,X
```

```
510    RTS
520 ;
530 PRNT
540    LDX #IOCB2
550    LDA #BUFLEN&255
560    STA ICBLL,X
570    LDA #BUFLEN/256
580    STA ICBLH,X
590    JSR CIOV
600    RTS
```

En las líneas 450 y 460 se almacena el número $0B, el símbolo de la operación "poner un carácter", en el byte de instrucción del IOCB #2. En líneas de la 480 a la 520, la dirección del buffer de texto que hemos creado especialmente para este programa se almacena en los bytes del buffer de dirección del IOCBC #2. Esto nos prepara para la rutina PRNT que comienza en la línea 540. En la rutina PRNT, que se extiende desde la línea 540 hasta la línea 610, la longitud de nuestro buffer de texto se almacena en los bytes de longitud del buffer del IOCB #2. Luego hay otro salto al vector CIO, que se encarga automáticamente de imprimir el texto del buffer de texto llamado PRNTSC en la pantalla del computador.

Cierre de un dispositivo

Cuando se abre un dispositivo en el lenguaje ensamblador (tal como en Atari BASIC), debe cerrarlo cuando haya terminado de usarlo. De lo contrario, se producirá un error IOCB, lo que podría causar algunos problemas graves.

El olvidar llevar a cabo tareas tales como cerrar los IOCBs (en el momento en que éstos deben ser cerrados) puede llevar a caídas de programas y sesiones de depuración largas y penosas. De todos modos, el IOCB #2 se cierra en esta versión del programa PRNTSC. En las Líneas de la 630 a la 680, el valor de $OC, el símbolo de cierre de archivo, se carga en ICCOM, X. Luego hay un salto a CIOV, y el sistema operativo de su Atari cierra el IOCB.

Esto termina nuestro breve vistazo a las complejidades del sistema central de entrada y salida de los computadores Atari. Pero de ninguna manera hemos acabado de ver este tema: hay mucha más información sobre E/S de su Atari en libros más avanzados que éste, y en los manuales de referencia técnica.

Capítulo Trece
Gráficos del Atari

Este capítulo y el que sigue son las ganancias reales en este libro sobre el lenguaje ensamblador. A continuación, aprenderá a utilizar el lenguaje ensamblador para:

- Diseñar sus propias pantallas, entremezclando texto, gráficos y colores de la manera que quiera.
- Desplazar (scroll) texto y gráficos en la pantalla del computador.
- Crear y usar sus propios conjuntos (sets) de caracteres.
- Utilizar los controles de juego (joysticks, paddles) en sus programas en lenguaje ensamblador.
- Usar gráficos player-missile de Atari para crear gráficos al estilo arcade en la pantalla de su computador.

Para llevar a cabo todas estas cosas, va a tener que escribir algunas rutinas en lenguaje ensamblador bastante complejas. Pero una vez que las haya digitado y guardado, verá que hay muchas maneras de usarlas. Al momento de terminar este libro, descubrirá (espero) que se ha convertido en un programador en lenguaje ensamblador bastante avanzado.

¡El primer programa de este capítulo le permitirá crear una pantalla de presentación que puede utilizar con cualquier programa hecho por usted, y que también llamará mucho la atención! Este programa despliega tres diferentes tamaños de letra en la pantalla del computador, con cada tamaño de un color diferente, y con un fondo de color distinto. Y, como dicen en los anuncios de televisión, "Pero aún hay más...". En el siguiente (y último) capítulo de este libro, aprenderá a usar desplazamiento fino (fine scrolling) para animar sus pantallas de presentación. Luego, como bono extra, aprenderá a utilizar los controles de juego y los gráficos de player-missile en sus programas en lenguaje ensamblador. Sin embargo, antes tendremos que echarle una breve mirada a los gráficos de su computador, y la forma en que éste genera su despliegue (display) por la pantalla.

El chip Antic

Los computadores Atari utilizan una técnica para generar el despliegue de la pantalla mucho más sofisticada que la utilizada por la mayoría de los computadores. En estos encontrará sólo unos bloques de memoria RAM en los que puede almacenar los datos que desea que aparezcan en la pantalla de su computador. En otras palabras, un bloque de memoria RAM que está dedicado a la memoria de la pantalla. Dentro de ese bloque de memoria RAM, a cada letra o símbolo que aparece en la pantalla se le asigna una posición de memoria. Cuando el valor de esa posición de memoria cambia, el texto o los gráficos que se encuentran en el lugar de la pantalla que corresponde a dicha posición de

memoria también cambiará. Y eso es todo lo que tiene que saber para entender el despliegue de gráficos y texto en la mayoría de los sistemas computacionales.

Como acabamos de decir, los gráficos de Atari son más sofisticados y también un poco más complicados. Este computador utiliza dos chips especiales para generar el despliegue de sus gráficos: el chip ANTIC y el GTIA. El primero es un microprocesador real, y está diseñado para ser utilizado con un conjunto de instrucciones especiales, y un tipo especial de programa llamado "lista de despliegue (display list)". Así que para poder crear gráficos con el chip Antic, tiene que saber cómo utilizar el conjunto de instrucciones para diseñar las listas de despliegue de su Atari. También ayuda el tener un conocimiento básico acerca de cómo funciona un aparato de televisión. Así que aquí va:

Líneas horizontales

Como usted sabe, la imagen de la pantalla de televisión se compone de pequeñas líneas horizontales: 262 líneas, para ser exactos. Y cada una de estas líneas se llama "línea horizontal (scan line)".

Como también debe saber, estas líneas horizontales son producidas por un cañón de electrones que se encuentra detrás del tubo de su aparato de televisión. Esta pistola electrónica dispara electrones al revestimiento de fósforo que se encuentra dentro del tubo del televisor en lo que se conoce como un "patrón de escaneo de raster - raster scan pattern", un patrón en zigzag que comienza en la esquina superior izquierda y termina en la esquina inferior derecha de la pantalla.

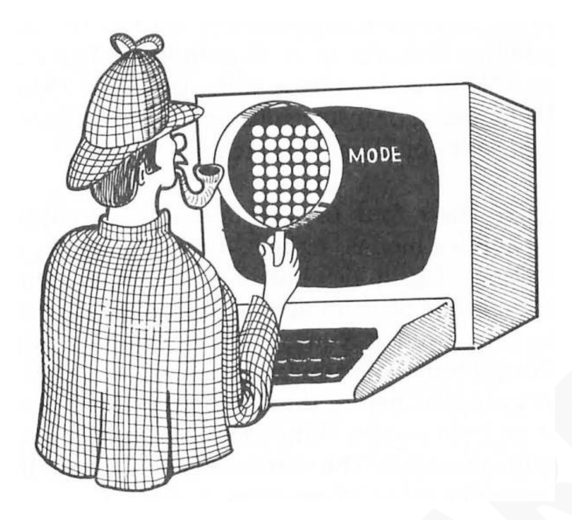
Las 262 líneas horizontales del televisor son reemplazadas 60 veces cada segundo por una imagen completamente nueva en la pantalla. Entre cada uno de estos rapidísimos cambios de escenario, hay un intervalo muy breve llamado intervalo de "borrado vertical - vertical blank", en la que toda la pantalla se queda en blanco.

Sin embargo, debido a una técnica de diseño de imagen llamada "overscan", no todas las 262 líneas horizontales están disponibles para una imagen en la pantalla de un televisor; algunas de estas líneas caen fuera del borde de la pantalla y por lo tanto nunca son vistas. Así que los programas utilizados para generar las imágenes de la pantalla de video de su computador no suelen hacer uso de todas estas líneas. Por ejemplo, su Atari utiliza sólo 192 de las 262 líneas horizontales disponibles.

Caracteres de matriz de puntos

Si mira con detalle el texto de una pantalla de TV generado por un computador, observará que cada carácter del texto en la pantalla se compone de pequeños puntos. Y si pudiera mirar con suficiente atención el texto de la pantalla generada por su Atari, mientras su equipo está en su modo normal de 40 columnas por 20 líneas de texto, podrá ver que

cada letra en la pantalla se compone de 64 puntos, dispuestos en una matriz de 8 puntos de ancho por 8 puntos de alto.



Líneas de modo

Su computador Atari tiene cuatro modos de texto diferentes. Cada uno de ellos genera caracteres de un tamaño distinto. Pero no importa lo grande que sean las letras en la pantalla, cada línea de texto en la pantalla de su Atari corresponde a una línea de modo. En el modo normal de 40 columnas por 24 líneas de texto, el modo conocido en el Atari BASIC como GRAPHICS 0, cada letra de una línea de modo tiene ocho puntos de alto, y cada uno de esos puntos equivale a una línea horizontal ("scan line"). Por lo tanto, una línea del modo GRAPHICS 0 del BASIC equivale a ocho líneas horizontales (scan lines).

El BASIC de Atari soporta otros dos modos de texto: el modo GRAPHICS 1, en el que los caracteres en la pantalla tienen la misma altura que los caracteres en el modo GRAPHICS 0, pero tienen el doble de ancho, y el modo GRAPHICS 2 en el que los caracteres tienen el doble de alto y el doble de ancho que los caracteres estándares del modo GRAPHICS 0. Cuando el computador está en modo GRAPHICS 1, cada línea del modo se compone de ocho líneas horizontales (scan lines), o sea, el mismo número de líneas horizontales utilizados en una línea de modo en GRAPHICS 0. Sin embargo, cuando el Atari está en modo GRAPHICS 2, cada línea de modo equivale a 16 líneas horizontales (scan lines).

Modo Antic

En el lenguaje ensamblador existe otro modo de texto, llamado Modo 3 del ANTIC, que no es compatible con BASIC. En el modo 3 del ANTIC, cada línea de modo está compuesta por 10 líneas horizontales (scan lines). Puede encontrar más información acerca del Modo 3 del ANTIC leyendo el manual del programador De Re Atari, o consultando el libro "The Atari 400/800 Technical Reference Notes" publicado por Atari.

Además de los cuatro modos de texto, los computadores Atari tienen numerosos modos gráficos, 10 o 13, dependiendo del tipo de hardware de gráficos que fue instalado en su Atari. (El número de modos gráficos ofrecidos por su computador Atari varían, ya que los modelos más viejos tienen un chip gráfico llamado CTIA, mientras que los nuevos vienen instalados con un chip GTIA nuevo). En los modos gráficos sin texto, el número de líneas horizontales (scan lines) por cada línea de modo puede varía entre uno (en gráficos de alta resolución) y ocho (en gráficos de baja resolución). El número de colores disponibles también cambia entre modos gráficos.

Dentro del modo

He aquí una tabla de los modos gráficos que se encuentran disponibles para los programadores de lenguaje ensamblador de Atari. Los lectores con ojo de águila notarán que existen diferencias entre los nombres ANTIC y BASIC de estos modos, y que el lenguaje ensamblador soporta varios modos que el BASIC de Atari no soporta. El cuadro no incluye los modos especiales disponibles para los dueños de los chips GTIA, ya que este libro es para todos los computadores personales Atari, y el programa que utiliza dichos modos no funciona correctamente en todos los computadores Atari. Si desea utilizarlo de todos modos, puede averiguar cómo hacerlo en la guía de programación en lenguaje ensamblador llamada De Re de Atari.

Modos de texto y gráficos del Atari

Modo ANTIC	Modo BASIC	Líneas horizontales por Línea de modo	Número de Colores
2	0	8	2
3	Ninguno	10	2
4	Ninguno	8	4
5	Ninguno	16	4
6	1	8	5
7	2	16	5
8	3	8	4
9	4	4	2
A	5	4	4
B	6	2	2
C	Ninguno	1	2
D	7	2	4
E	Ninguno	1	4
F	8	1	2

Personalizando la pantalla de su Atari

Se necesitan dos etapas para personalizar la pantalla de su Atari. Primero tiene que crear un tipo especial de programa que se llama "lista de visualización (o despliegue) - display list". Luego tiene que escribir un programa que le dirá a su equipo cómo utilizar la lista de

despliegue que acaba de diseñar. Mientras trabaje en esto, tenga presente que una lista de despliegue no puede sobrepasar el límite de 1K y que la memoria de una lista de despliegue no puede sobrepasar los 4K sin hacer un manejo especial.



En un momento hablaremos acerca de cómo escribir un programa de lista de despliegue. Sin embargo, antes echaremos un vistazo a la lista de despliegue que se referirá el programa ANTIC. Una lista de despliegue se compone de una serie de instrucciones de un byte que se pueden colocar casi en cualquier lugar disponible de la memoria RAM de su computador. Para tener una idea de cómo se ve una lista de despliegue, puede utilizar la herramienta de depuración de su ensamblador para examinar la lista de despliegue que utiliza su equipo cuando está en su modo de texto GRAPHICS 0.

Al encender su computador, este pasa automáticamente al modo GRAPHICS 0, y la dirección de la lista de despliegue que se utiliza para generar ese modo siempre se almacena en dos lugares: la dirección de memoria $230 y $231. La dirección de memoria $230 siempre guarda el byte bajo de la dirección donde comienza la lista de despliegue que su computador está usando, y la dirección de memoria $231 siempre guarda el byte alto de la dirección de comienzo de la lista de despliegue. Así que una vez que conozca el contenido de estas dos direcciones, podrá encontrar la lista de despliegue que su computador está utilizando actualmente. Una vez que localice la lista de despliegue del modo GRAPHICS 0 de su equipo, verá que esta se ve algo así:

```
70   70   70   42   20   7C   02   02
02   02   02   02   02   02   02   02
02   02   02   02   02   02   02   02
02   02   02   02   02   41   E0   7B
```

Como puede ver, una lista de despliegue es simplemente eso: una lista, no un programa. Para utilizar una lista de despliegue, se necesita otro programa. Tendrá la oportunidad de

echar un vistazo a dicho programa en un momento. Pero antes vamos a examinar byte por byte este ejemplo de lista de despliegue.

Bytes 1 al 3

$70 $70 $70

Cada byte en una lista de despliegue tiene un significado específico para el chip ANTIC de Atari. Y dentro de cada byte, cada nibble (esto es cada dígito hexadecimal) también tiene un significado específico. Por ejemplo, cada uno de los tres primeros bytes - cada uno de los $70 que aparecen al principio de la lista - le dice al chip ANTIC que despliegue una línea de modo en blanco (o sea en GRAPHICS 0 del BASIC, ocho líneas horizontales (scan lines) en blanco). Una lista de despliegue estándar en GRAPHICS 0 siempre comienza con tres instrucciones de "salto de línea de modo" (en el lenguaje ANTIC, tres $70) a fin de superar las características del overscan de las TVs y asegurarse de que la visualización completa que se pide en la lista de despliegue sea visible completamente en la pantalla.

Bytes 4 al 6

$42 $20 $7C

El primer byte efectivo de nuestra lista de despliegue de ejemplo ($42) es lo que se conoce como un comando de "Carga de Memoria de Escaneo - Load Memory Scan (LMS)". El primer byte a desplegar en una lista de despliegue, es decir, el primer byte después de todas las líneas en blanco necesarias, siempre es un comando LMS. Y el comando de carga memoria de escaneo siempre tiene tres bytes. En la lista de despliegue que ahora nos ocupa, el comando de carga de memoria de escaneo se compone de tres bytes: "$ 42 $ 20 $ 7C." El primer nibble de esta instrucción, el número 4, alerta al ANTIC que lo que viene a continuación es una instrucción LMS. El segundo nibble en la instrucción LMS, el dígito 2, le dice a ANTIC que muestre una línea en modo ANTIC 2. Consulte la tabla de modos gráficos presentada algunos párrafos atrás, y verá que en el lenguaje Antic, el modo 2 es el mismo que el modo 0 de BASIC. Los siguientes dos bytes del comando LMS, $20 $7C, proporcionan a ANTIC la dirección de memoria en la que comenzará la pantalla. ANTIC interpreta estos dos bytes partiendo por el byte bajo, como es la norma en el 6502. Por lo tanto, cuando ANTIC se encuentra con la instrucción LMS $42 $20 $7C, el primer byte que aparece en la pantalla de video de su Atari será el byte que está almacenado en la dirección de memoria $7C20.

Cuando escribe una lista de despliegue, puede poner su memoria de pantalla en cualquier bloque de memoria RAM que sea conveniente y que se encuentre disponible. Y puede llenar esa memoria RAM con lo que quiera: los códigos ATASCII que equivalen a un trozo de texto, una pantalla o caracteres gráficos elaborados con la ayuda de un programa de gráficos. Una vez que haya creado una lista de despliegue, ponga la dirección de la pantalla en los dos bytes que siguen la orden LMS de su lista de despliegue. Tendrá la

oportunidad de ver cómo trabaja esta técnica en el programa de ejemplo que se encuentra al final de este capítulo.

Bytes 7 - 29

El byte $02, repetido 23 veces

Como se explicó anteriormente, el primer comando LMS de una lista de despliegue le dice a ANTIC dos cosas: la dirección en que comienza la memoria de pantalla, y el modo gráfico a utilizar para mostrar la primera línea de texto o datos que se encuentra a partir de esa dirección. Después de que ANTIC cuente con esta información, se le debe indicar qué modo gráfico tiene que utilizar para mostrar cada "línea de modo" que se desplegará posteriormente en la pantalla. En la lista de visualización que ahora estamos examinando, cada "línea de modo" en la pantalla es una línea de Modo ANTIC 2. Por lo tanto, las siguientes 23 instrucciones de esta lista de despliegue son todas iguales: Cada una de ellas le dice a ANTIC que la siguiente línea de la pantalla será una línea en Modo ANTIC 2.

Usted se pregunta: ¿Qué pasaría si todas estas 23 instrucciones no fueran iguales? Bueno, si fueran distintas entonces se puede mostrar de manera simultánea más de un modo gráfico en la pantalla. Se puede mostrar texto de distintos tamaños, en la misma pantalla, y se pueden combinar modos de texto y modos gráficos como usted lo desee. Esta es una muy potente e inusual capacidad de los computadores Atari. Tendrá la oportunidad de ver exactamente cómo funciona esto antes de terminar este capítulo.

Bytes del 30 al 32

$41 $E0 $7B

Cada lista de despliegue debe terminar con una orden de tres bytes llamada instrucción JVB (Salto en el barrido vertical - Jump on Vertical Blank). El primer byte de una instrucción de VB siempre es $41. Los próximos dos bytes siempre se combinan para formar una dirección de salto. El destino del salto es siempre el principio de la lista de despliegue en el salto está contenido. De hecho, la lista de despliegue que estamos examinando comienza en la dirección de memoria $7BE0. Así que esa es la dirección, comenzando por el byte bajo, que viene a continuación de la instrucción $41 de JVB. Cuando ANTIC encuentra la instrucción $41 de JVB en una lista de despliegue, salta de nuevo al principio de la lista, espera el siguiente período de barrido vertical que se da entre imágenes, y luego salta a la dirección que viene a continuación de la instrucción JVB. Ya que esta dirección corresponde el inicio de la lista de despliegue, lo que la instrucción JVB realmente hace es generar la lista de despliegue de nuevo.

La ejecución de una lista de despliegue

Como hemos señalado, una lista de despliegue se puede colocar en casi cualquier punto disponible y conveniente de la memoria de su computador. De la misma manera, la

memoria de la pantalla se puede colocar casi en cualquier lugar de la memoria RAM. Una vez que haya creado una lista de despliegue y un bloque de datos que se utilizará como memoria de pantalla, todo lo que tiene que hacer para desplegar su diseño personalizado de pantalla es escribir un pequeño y simple programa en lenguaje ensamblador que le indica al sistema operativo de su computador dónde se encuentra su lista de despliegue. Para indicarle a su computador dónde se encuentra la lista de despliegue personalizada, todo lo que tiene que hacer es almacenar los nuevos valores en un par de ubicaciones de memoria del sistema operativo conocidas como las posiciones "shadow (sombra)". Las direcciones "shadow (sombra)" se utilizan a menudo en la programación de su Atari, por lo que le explicaré ahora de qué se tratan.

En la memoria de su computador hay algunos registros de hardware muy útiles que normalmente no pueden ser accedidos por los programas escritos por los usuarios. Sin embargo, sesenta veces por segundo, los datos de cada una de estas posiciones de memoria son actualizadas. Durante este proceso de actualización, el valor almacenado en cada uno de estos registros se sustituye por los datos que se han almacenado en su registro "shadow (sombra)" correspondiente. Y los registros sombra están en la memoria RAM que puede ser accedida por el usuario. Así, al cambiar el valor de un registro sombra, puede cambiar el valor del correspondiente registro hardware. Por lo tanto, para la mayoría de los intentos y propósitos, un registro sombra trabaja casi igual que cualquier otro registro del sistema operativo que se encuentre en la memoria RAM. Tres direcciones sombra que se utilizan a menudo en los programas de listas de despliegue son $22F, $230 y $231. La dirección $22F es una ubicación de memoria del SO Atari llamada SDMCTL (“Shadow, Direct Memory Access Control - Sombra, Control de Acceso Directo a Memoria”). Las direcciones $230 y $231 son posiciones del OS llamadas SDLSTL (“Shadow, Display List Pointer - Low, Sombra, Puntero de la Lista de Despliegue – Bajo”) y SDLSTH (“Shadow, Display List Pointer - High, Sombra, Puntero de la Lista de Despliegue – Alto”). Para escribir un programa que pondrá una lista de despliegue personalizada en la pantalla de su Atari, todo lo que tiene que hacer es seguir estos tres pasos:

1. Apague su chip ANTIC almacenando un cero en la dirección $22F (SDMCTL).

2. Almacene la dirección de comienzo de su lista de despliegue personalizada en las direcciones $230 y $231 (SDLSTL y SDLSTH).

3. Encienda nuevamente su chip ANTIC almacenando un valor $22 en la dirección $22F (SDMCTL).

Ejecutándolo

Ahora que sabe cómo se hacen todas esas cosas, estamos listos para comenzar. He aquí una lista de despliegue personalizada junto con un programa que la ejecutará. Si lo

desea, puede digitarlo en su computador Atari, guardarlo en un disco, y ejecutarlo ahora mismo:

UNA PANTALLA PERSONALIZADA

```
10   ;
20   ; PANTALLA HOLA
30   ;
40     *=$3000
50     JMP INIT
60   ;
70   SDMCTL=$022F
80   ;
90   SDLSTL=$0230
100  SDLSTH=$0231
110  ;
120  COLOR0=$02C4; REGISTRO DE COLOR S.O.
130  COLOR1=$02C5
140  COLOR2=$02C6
150  COLOR3=$02C7
160  COLOR4=$02C8
170  ;
180  ;DATOS DE LA LISTA DE DESPLIEGUE
190  ;
200  START
210  ;
220  LINE1 .SBYTE "    PRESENTANDO     "
230  LINE2 .SBYTE "  el gran programa  "
240  LINE3 .SBYTE "   Por (su nombre)  "
250  ;
260  LINE4 .SBYTE "  POR FAVOR ESPERE  "
270  ;
280  ;LISTA DE DESPLIEGUE
290  ;
300  HLIST
310     .BYTE $70,$70,$70; 3 LINEAS EN BLANCO
320     .BYTE $70,$70,$70,$70,$70; MAS LINEAS EN BLANCO
330     .BYTE $46; LSM, MODO ANTIC 6 (MODO BASIC 2)
340     .WORD LINE1; (LINEA DE TEXTO: "PRESENTANDO...")
350     .BYTE $70,$70,$70,$70,$47; LMS, MODO ANTIC 7
360     .WORD LINE2; (LINEA DE TEXTO: "EL GRAN PROGRAMA")
370     .BYTE $70,$42; (LMS, MODO ANTIC 2 [GR.0])
380     .WORD LINE3;( LINEA DE TEXTO: "Por [su nombre]")
390     .BYTE $70,$70,$70,$70,$46;LMS, MODO ANTIC 6
400     .WORD LINE4; (LINEA DE TEXTO: "POR FAVOR ESPERE")
410     .BYTE $70,$70,$70,$70,$70; 5 LINEAS EN BLANCO
420     .BYTE $41; INSTRUCCION JVB
430     .WORD HLIST; SALTAR DE NUEVO AL COMIENZO DE LA LISTA
440  ;
450  ;EJECUTAR PROGRAMA
460  ;
```

```
470 INIT; INTERCAMBIO DE REGISTROS DE COLOR PARA UN BONITO
    DESPLIEGUE DE COLORES
480    LDA COLOR3
490    STA COLOR1
500    LDA COLOR4
510    STA COLOR2
520 ;AHORA EJECUTAREMOS EL PROGRAMA
530    LDA #0
540    STA SDMCTL; APAGAMOS ANTIC POR UN MOMENTO
550    LDA #HLIST&255; MIENTRAS ALMACENAMOS LA DIRECCION DE
       NUESTRA NUEVA LISTA
560    STA SDLSTL; EN EL PUNTERO DE DESPLIEGUE DE S.O.
570    LDA #HLIST/256; AHORA EL BYTE ALTO
580    STA SDLSTH; AHORA ANTIC SABRA NUESTRA NUEVA DIRECCION
590    LDA #$22
600    STA SDMCTL; ...ASI QUE AHORA ENCENDEMOS ANTIC
610 ;
620 FINI
630    RTS
```

Cómo funciona

Ya hemos visto prácticamente todo lo que aparece en este programa, por lo que se explica bastante bien por sí mismo. Sin embargo, si está utilizando un cartucho Atari Assembler o el paquete Atari Macro Assembler and Text Editor, tendrá un problema con este programa: ¡no funcionará! Esto se debe a que ambos ensambladores de Atari, a pesar de ser tan sofisticados, no están equipados con una pequeña pero útil directiva que tiene el ensamblador MAC/65. Esta es la directiva .SBYTE que se usa en las líneas de la 220 a la 260 de nuestro programa de lista de despliegue.

La directiva .SBYTE del MAC/65 fue diseñada para convertir de código ATASCII, que es el que usa su computador para almacenar texto en la memoria, a un código completamente diferente que se utiliza para mostrar los caracteres en la pantalla. Este último código se llama, muy apropiadamente, "código de pantalla - screen code". No sería difícil convertir de ATASCII a código de pantalla si hubiera una correlación directa de uno a uno entre ambos códigos. Pero desafortunadamente no existe tal relación. Para traducir del código ATASCII al código de pantalla, hay que sumar 64 a algunos códigos de caracteres, restar 32 a otros, y dejar otros sin sumar nada. He aquí una tabla de conversión de código ATASCII a código de pantalla que muestra exactamente lo que implica este proceso de traducción:

Convertir de ATASCII a Código de Pantalla

VALORES ATASCII	OPERACION NECESARIA PARA LA CONVERSION
0 al 31	Sumar 64
32 al 95	Restar 32
96 al 127	Ninguna
128 al 159	Sumar 64
160 al 223	Restar 32
224 al 255	Ninguna

Una rutina de conversión automática

Si su ensamblador no tiene una función .SBYTE, puede utilizar esta rutina en lenguaje ensamblador para hacer las conversiones descritas anteriormente. La escribí antes de que existiera el ensamblador MAC/65 o la directiva .SBYTE. Es una rutina "rápida y sucia" que fue concebida en un apuro, y si es un buen programador en lenguaje ensamblador, probablemente podrá escribir una rutina que hace el mismo trabajo de manera más eficiente, o más rápida, o ambas. Pero funciona bien, y con un bloque de texto de cualquier tamaño. Esta pequeña subrutina de conversión simplemente realiza los cálculos de la tabla anterior de manera automática. Sin embargo, antes de poder usarla, tendrá que hacer algunas modificaciones menores en el programa de lista de despliegue que escribió hace unos momentos. Tendrá que hacer lo siguiente:

Tres modificaciones

- Convertir las directivas .SBYTE de las líneas de la 220 a la 260 a directivas BYTE (o directivas DB, si está utilizando el Macro Assembler de Atari).
- Agregar una variable y una constante a la tabla de símbolos de su programa de lista de despliegue. La variable, que llamaremos TEMPTR (por "temporary pointer - puntero temporal"), tendrá que estar ubicada en la página cero, ya que será utilizada con direccionamiento indirecto indexado, un modo de direccionamiento que exige dos posiciones en la página cero. La constante que va a utilizar, denominada EOF, tendrá el valor literal $88, que es el código ATASCII del carácter "fin de archivo". Puede agregar estos símbolos a la tabla de símbolos de su programa de títulos por pantalla con estas líneas:

```
65 TEMPTR = $CC
66 EOF = $88
67 ;
```

- Agregar esta línea a su programa para marcar el final del bloque de texto que será convertido a código de pantalla:

```
265  .BYTE EOF
```

Ahora comencemos

La rutina de conversión comienza almacenando la dirección inicial de los datos de nuestra nueva lista de despliegue en el par de punteros llamados TEMPTR y TEMPTR+1. Luego, usando direccionamiento indirecto indexado, se desplaza por el texto que será convertido carácter por carácter. Sin embargo, antes de que se realice cada conversión, se comprueba si el carácter en cuestión es un carácter de fin de archivo ($ 88). Si el carácter en cuestión es un carácter EOF, termina la subrutina, ya que significa que se han realizado todas las conversiones necesarias y que es el momento de terminar la rutina de conversión. Si el carácter no es un carácter EOF ("end of file - fin de archivo"), el programa sigue adelante y realiza la conversión necesaria (si es necesario). Luego se pasa al siguiente carácter.

Imitando la Directiva. SBYTE

Si su ensamblador no tiene una directiva .SBYTE, o si, por alguna razón, no desea utilizar la directiva .SBYTE que trae el ensamblador MAC/65, entonces puede utilizar esta rutina de conversión. Sólo tiene que digitarla en la memoria RAM, y luego saltar a ésta después de que su lista de despliegue se cargue en memoria, pero antes de que comience el código que inicializa su lista de despliegue. Puede hacer esto con la siguiente línea:

```
475  JSR FIX
```

He aquí el programa de conversión:

Nota: Antes de ensamblar este programa use el comando SIZE para verificar que MEMLO (el segundo número) es menor que la dirección inicial del código de objeto en la línea 40. Si no es así, debe ya sea cambiar la dirección de comienzo (por ejemplo, 40 *= $ 5000), o cambiar LOMEM (por ejemplo, LOMEM 4000), o eliminar los comentarios del código fuente para que MEMLO sea menor que la dirección de comienzo.

Una subrutina de conversión de ATASCII a ASCII

```
2000 ;
2010 ;ARREGLA DATOS
2020 ;
2030 FIX
2040    LDA #START&255; DIRECCION DE COMIENZO DE LA NUEVA LISTA
        DE DESPLIEGUE (BYTE BAJO)
2050    STA TEMPTR
2060    LDA  #START/256;  NUEVA  DIRECCION  DE  LA  LISTA  DE
        DESPLIEGUE (BYTE ALTO)
2070    STA TEMPTR+1
```

```
2080 ;
2090 ; SUBRUTINA "ARREGLA DATOS"
2100 ;
2110    LDY #0; CARGAR EL REGISTRO Y CON 0 PARA DIRECCIONAMIENTO INDIRECTO INDEXADO
2120 FXDT
2130    LDA (TEMPTR),Y; COMIENZA CON EL PRIMER CARACTER DEL BLOQUE
2140    CMP #EOF; ES UN CARACTER EOF($88)?
2150    BEQ DONE; SI ES ASI, TERMINA - - SALIR DE LA SUBRUTINA
2160    JSR FXCH; SI NO, SALTE A AL SUBRUTINA "ARREGLA CARACTER"
2170    STA (TEMPTR), Y; LUEGO REEMPLACE EL CARACTER VIEJO CON EL NUEVO
2180    INC TEMPTR; ...E INCREMENTE TEMPTR (BYTE BAJO)
2190    BNE FXDT; SI NO HAY ACARREO AL BYTE ALTO, COMIENCE OTRA VEZ
2200    INC TEMPTR+1; SI NO, INCREMENTE EL BYTE ALTO DE TEMPTR
2210    JMP FXDT; ...Y LUEGO VUELVA A FXDT Y COMIENCE OTRA VEZ
2220 ;
2230 DONE; FIN DE LA SUBRUTINA PRINCIPAL - - TODOS LOS CARACTERES HAN SIDO CONVERTIDOS
2240    RTS; ASI QUE VOLVEMOS AL PROGRAMA EN CURSO
2250 ;
2260 ; RUTINA "ARREGLA CARACTER"
2270 ;
2280 FXCH
2290    CMP #32; EL CODIGO DE CARACTER ES < 32?
2300    BCC ADD64; SI LO ES, SALTE A RUTINA "ADD64"
2310 ;
2320    CMP #96; ES < 96?
2330    BCC SUB32; SI LO ES, SALTE A RUTINA "SUB32"
2340 ;
2350    CMP #128; < 128?
2360    BCC FIXT; SI LO ES; SALTE A RUTINA "FIXT" (SIN ACCION)
2370 ;
2380    CMP #160; < 160?
2390    BCC ADD64; SI LO ES, SALTE A ADD64
2400 ;
2410    CMP #224; < 224?
2420    BCC SUB32; SI LO ES, SALTE A SUB32
2430 ;
2440    JMP FIXT; SI ESTA ENTRE 224 Y 255, NO SE HACE NADA
2450 ;
2460 ADD64
2470    CLC; LIMPIAR ACARREO PARA SUMAR
2480    ADC #64; LUEGO SUMA 64
2490    JMP FIXT; REGRESA A RUTINA PRINCIPAL (FXDT)
2500 ;
2510 SUB32
2520    SEC; ACTIVAR ACARREO PARA RESTAR
2530    SBC #32; Y RESTAR 32
```

```
2540  FIXT
2550     RTS; REGRESAR A SUBRUTINA PRINCIPAL (FIX)
```

Desplazamiento grueso

Puede ejecutar este programa tan pronto como lo haya escrito y lo haya guardado en disco, y si lo digitó tal como está escrito, no debería tener problemas. En cuanto lo tenga funcionando, puede comenzar a arreglarlo de modo que sea aún más elegante, con una técnica conocida como el desplazamiento (scrolling). Sin embargo, antes de iniciar un debate sobre la técnica de desplazamiento, me gustaría sugerirle que haga otra pequeña modificación al programa de lista de despliegue que hemos estado viendo. Desplazaremos sólo una línea del programa - la línea que dice "EL GRAN PROGRAMA". Y tendremos que insertar algunos espacios en blanco antes y después de esa línea de modo que se desplace correctamente por la pantalla. Con estos espacios insertados, la pantalla comenzará con espacios en blanco donde deberían estar las palabras "EL GRAN PROGRAMA". Entonces, estas tres palabras se desplazarán por la pantalla, como esos letreros que parecen cintas de teletipo que a veces se ven en las tiendas y en la televisión. Para hacer este nuevo espacio, tendrá que cambiar una línea de su programa original de lista de despliegue y, a continuación, agregar dos líneas más. Aquí están las tres líneas que necesitaremos (la 230 es la línea modificada, y la 225 y la 235 son las nuevas):

```
225   LINE2 .SBYTE "                         "
230     .SBYTE "     el gran programa    "
235     .SBYTE "                         "
```

Una vez que se modifiquen estas líneas, no será difícil implementar un tipo de desplazamiento primitivo (llamado desplazamiento grueso) en nuestro programa de lista de despliegue. Para implementar un desplazamiento grueso, todo lo que tiene que hacer es crear un ciclo que aumente o disminuya ciertas direcciones del programa - en concreto, las direcciones que siguen a las instrucciones LMS en una lista de despliegue. Si su programa de desplazamiento está escrito en lenguaje ensamblador, también tendrá que incluir algún tipo de ciclo de retardo, ya que el lenguaje de máquina es tan rápido que sin ningún tipo de retraso, la línea de desplazamiento pasará tan rápido por la pantalla que se convertirá en un borrón.

Aquí está la lista de despliegue y una rutina de implementación de lista de despliegue que agregará el desplazamiento grueso a su programa de títulos en la pantalla. Realice los siguientes cambios en el programa, y la línea que dice "ESTE GRAN PROGRAMA" se desplazará por la pantalla una y otra vez en un ciclo sin fin. El texto se mueve de una manera muy tosca, una letra a la vez. Pero no se preocupe, en el próximo capítulo, vamos a suavizar esta acción con una técnica de lenguaje ensamblador conocida como desplazamiento fino.

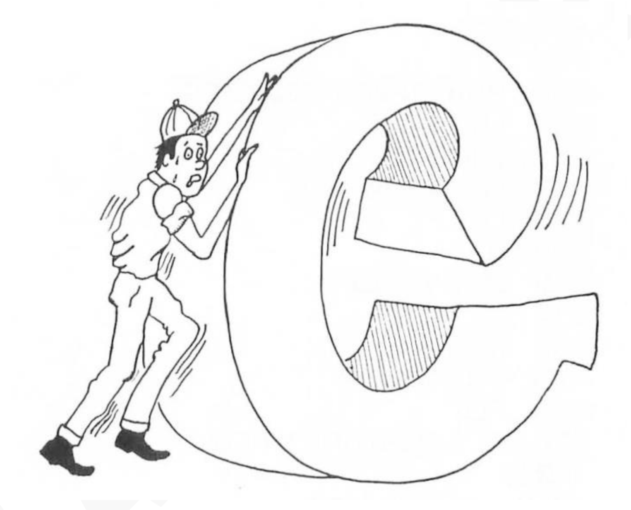
Un ejemplo de desplazamiento grueso

```
10    ;
20    ; PANTALLA HOLA (GRUESO)
30    ;
40       *=$3000
50       JMP INIT
60    ;
70    TCKPTR=$2000
80    ;
90    SDMCTL=$022F
100   ;
110   SDLSTL=$0230
120   SDLSTH=$0231
130   ;
140   COLOR0=$02C4; REGISTOS DE COLOR DEL S.O.
150   COLOR1=$02C5
160   COLOR2=$02C6
170   COLOR3=$02C7
180   COLOR4=$02C8
190   ;
200   ;DATOS DE LA LISTA DE DESPLIEQUE
210   ;
220   START
230   LINE1 .SBYTE "   PRESENTANDO A   "
240   LINE2 .SBYTE "                      "
250      .SBYTE "   EL GRAN PROGRAMA   "
260      .SBYTE "                      "
270   LINE3 .SBYTE "             Por (su"
280      .SBYTE "nombre)               "
290   LINE4 .SBYTE "   POR FAVOR ESPERE"
300   ;
310   ;LISTA DE DESPLIEGUE DE SALUDO
320   ;
330   HLIST
340      .BYTE $70,$70,$70
350      .BYTE $70,$70,$70,$70,$70
360      .BYTE $46
370      .WORD LINE1
380      .BYTE $70,$70,$70,$70,$47
390   SCROLN
400      .WORD $00
410      .BYTE $70,$42
420      .WORD LINE3
430      .BYTE $70,$70,$70,$70,$46
440      .WORD LINE4
450      .BYTE $70,$70,$70,$70,$70
460      .BYTE $41
470      .WORD HLIST
480   ;
490   ;EJECUCION DEL PROGRAMA
```

```
500   ;
510   INIT
520      LDA COLOR3
530      STA COLOR1
540      LDA COLOR4
550      STA COLOR2
560   ;
570      LDA #0
580      STA SDMCTL
590      LDA #HLIST&255
600      STA SDLSTL
610      LDA #HLIST/256
620      STA SDLSTH
630      LDA #$22
640      STA SDMCTL
650   ;
660   ;RUTINA DE DESPLAZAMIENTO GRUESO
670   ;
680      LDA #40
690      STA TCKPTR
700      JSR TCKSET
710   ;
720   COARSE
730      LDY TCKPTR; 40 PARA COMENZAR
740      DEY
750      BNE SCORSE
760      LDY #40
770      JSR TCKSET
780   SCORSE
790      STY TCKPTR
800      INC SCROLN
810      BNE LEAP
820      INC SCROLN+1
830   ;
840   ;CICLO DE RETRASO
850   ;
860   LEAP
870      TYA
880      PHA; GUARDAR REGISTRO Y
890      LDX #$FF
900   XLOOP
910      LDY #$80
920   YLOOP
930      DEY
940      BNE YLOOP
950   ;
960      DEX
970      BNE XLOOP
980      PLA
990      TAY; REESTABLECER REGISTRO Y
1000  ;
1010     JMP COARSE
```

```
1020 ;
1030 TCKSET
1040    LDA #LINE2&255
1050    STA SCROLN
1060    LDA #LINE2/256
1070    STA SCROLN+1
1080 ENDIT
1090    RTS
```

Si escribe las modificaciones antes mencionadas en su programa de lista de despliegue y lo ejecuta, lo que verá es un ejemplo de desplazamiento grueso. La línea que dice: "EL GRAN PROGRAMA" vendrá saltando a través de la pantalla, una letra a la vez, en una especie de forma desigual que podría llegar a ser más desconcertante aún si tiene que mirar la pantalla por mucho tiempo. El desplazamiento grueso es bastante malo cuando se usa para mover una sola línea a través de una pantalla. Pero se vuelve aún más enervante cuando se utiliza para desplazar una pantalla completa.



Para desplazar más de una línea en la pantalla, hasta e incluyendo el número máximo de líneas en una pantalla completa, todo lo que tiene que hacer es ir a su lista de despliegue e insertar una instrucción LMS antes de cada línea que desea desplazar. Si desea desplazar toda la pantalla, ¡puede preceder cada línea con una instrucción LMS! Luego, para desplazar la pantalla de manera horizontal, todo lo que tiene que hacer es crear un ciclo que progresivamente incremente o decremente la dirección inicial de los datos que aparecen en cada línea. Si incrementa dichas direcciones, la pantalla se desplazará de derecha a izquierda. Si las decrementa, se desplazará de izquierda a derecha.

Hacer desplazamiento vertical grueso es tan fácil como hacer desplazamiento horizontal grueso. Para desplazar una pantalla de manera vertical, todo lo que tiene que hacer es incrementar o disminuir la dirección LMS de cada línea el número de caracteres de las líneas que se desean desplazar, en lugar de un solo carácter a la vez. Al configurar este tipo de desplazamiento, tiene que contar cuidadosamente el número de caracteres de cada línea, de modo que sus caracteres no se muevan hacia adelante y atrás en la

pantalla a medida que se desplazan hacia arriba o hacia abajo. El desplazamiento vertical grueso, tal como el desplazamiento horizontal grueso, puede ser utilizado para desplazar cualquier número de líneas en la pantalla del monitor. De hecho, ya sea que esté consciente de ello o no, probablemente se ha topado muchas veces con el desplazamiento vertical grueso en acción. Es el tipo de desplazamiento que se ve cuando se está escribiendo una programa BASIC o en lenguaje ensamblador, al llegar a la línea inferior de la pantalla, y al presionar la tecla RETURN. Cuando se hace eso, la pantalla se mueve completa una línea hacia arriba, usando una rutina de desplazamiento vertical grueso.

El desplazamiento grueso funciona muy bien cuando se usa de esta manera (sólo una línea a la vez) pero no funciona muy bien en aplicaciones más exigentes, tales como el movimiento de pantallas de juegos al estilo arcade. Por desgracia para los programadores BASIC, el desplazamiento grueso es el único que admite el BASIC de Atari. Para hacer el desplazamiento suave, tiene que utilizar - ¡adivinó! - el lenguaje ensamblador. En el siguiente (y último capítulo) de este libro, aprenderá cómo usar el desplazamiento suave en programas en lenguaje ensamblador. Y como si eso no fuera suficiente, también aprenderá a crear y animar personajes de juegos, cómo utilizar gráficos player/missile, y cómo escribir programas en lenguaje ensamblador que requieran el uso de controles de juego (joysticks).

Capítulo Catorce
Gráficos Avanzados en el Atari

No importa qué tan buen programador sea, siempre habrá cosas que no se podrán hacer por medio de un programa BASIC. Es que BASIC no es lo suficientemente rápido para ocuparse de tareas como el desplazamiento fino, animación de caracteres a alta velocidad, y gráficos player/missile. El lenguaje ensamblador puede manejar estas tres tareas con facilidad, y en este capítulo verá exactamente cómo lo hace. Vamos a empezar con el desplazamiento fino (fine scrolling).



Desplazamiento Fino

Para demostrar el desplazamiento fino vamos a utilizar una versión ampliada del programa de pantallas de títulos que digitó en el capítulo anterior. Si guardó ese programa en un disco, cárguelo en su equipo ahora mismo. Luego, editándolo un poco y agregándole algunas cosas podrá modificarlo hasta que se parezca al listado de abajo. Luego de que lo haya modificado, guárdelo en un disco, ejecútelo, y échele un vistazo mientras se ejecuta. A continuación explicaré cómo funciona, y usted tendrá un llamativo programa en lenguaje ensamblador que le ayudará de ahora en adelante a crear pantallas de título para sus propios programas.

Una demostración de Desplazamiento Fino

```
10    ;
20    ;PANTALLA HOLA (FINO)
30    ;
40      *=$3000
50      JMP INIT
60    ;
70    TCKPTR=$2000
80    FSCPTR=TCKPTR+1
90    ;
100   SDMCTL=$022F
110   ;
120   SDLSTL=$0230
130   SDLSTH=$0231
140   ;
150   COLOR0=$02C4; REGISTROS DE COLOR DEL S.O.
160   COLOR1=$02C5
170   COLOR2=$02C6
180   COLOR3=$02C7
190   COLOR4=$02C8
200   ;
210   HSCROL=$D404
220   ;
230   VVBLKI=$0222; VECTOR DE INTERRUPCION DEL S.O.
240   SYSVBV=$E45F; VECTOR DE ACTIVACION DE LA INTERRUPCION
250   ;
260   SETVBV=$E45C; ACTIVACION DEL VECTOR DE LA INTERRUPCION DEL
      BORRADO VERTICAL
270   XITVBV=$E462; SALE DEL VECTOR VBI
280   ;
290   ; DATOS DE LA LISTA DE DESPLIEGUE
300   ;
310   START
320   LINE1 .SBYTE " PRESENTANDO A  "
330   LINE2 .SBYTE "                "
340     .SBYTE " el gran programa  "
350     .SBYTE "                   "
360   LINE3 .SBYTE "          Por (Su"
370     .SBYTE " Nombre)           "
380   LINE4 .SBYTE "POR FAVOR ESPERE "
390   ;
400   ; LISTA DE DESPLIEGUE CON LINEA DESPLAZANDOSE
410   ;
420   HLIST ;('HOLA' LIST)
430     .BYTE $70,$70,$70
440     .BYTE $70,$70,$70,$70,$70
450     .BYTE $46
460     .WORD LINE1
470   ;FIJESE QUEEL ULTIMO BYTE EN LA
480   ;SIGUIENTE LINEA ES $57, Y NO $47 COMO
```

```
490  ;LO ERA EN EL CAPITULO ANTERIOR
500    .BYTE $70,$70,$70,$70,$57
510  SCROLN ;(ESTA ES LA LINEA QUE DESPLAZAREMOS)
520    .WORD $00; UN ESPACIO QUE SERA LLENADO DESPUES
530    .BYTE $70,$42
540    .WORD LINE3
550    .BYTE $70,$70,$70,$70,$46
560    .WORD LINE4
570    .BYTE $70,$70,$70,$70,$70
580    .BYTE $41
590    .WORD HLIST
600  ;
610  ;EJECUTAR PROGRAMA
620  ;
630  INIT ;PREPARESE A EJECUTAR EL PROGRAMA
640    LDA COLOR3; ACTIVAR REGISTROS DE COLOR
650    STA COLOR1;
660    LDA COLOR4;
670    STA COLOR2;
680  ;
690    LDA #0
700    STA SDMCTL
710    LDA #HLIST&255
720    STA SDLSTL
730    LDA #HLIST/256
740    STA SDLSTH
750    LDA #$22
760    STA SDMCTL
770  ;
780    JSR TCKSET; INICIALIZAR DIRECCION DEL CRONOMETRO
790  ;
800    LDA #40; NUMERO DE CARACTERES EN LA LINEA DE
       DESPLAZAMIENTO
810    STA TCKPTR
820    LDA #8
830    STA FSCPTR; NUMERO DE RELOJES DE COLOR PARA
       DESPLAZAMIENTO FINO
840  ;
850  ; ACTIVAR INTERRUPCION
860  ;
870    LDY #TCKINT&255
880    LDX #TCKINT/256
890    LDA #6
900    JSR SETVBV
910  ;
920  ; INTERRUPCION DEL CRONOMETRO
930  ;
940  TCKINT
950    LDA #SCROLL&255
960    STA VVBLKI
970    LDA #SCROLL/256
980    STA VVBLKI+1
```

```
990   ;
1000  INFIN
1010    JMP INFIN; CICLO INFINITO
1020  ;
1030  SCROLL
1040    LDX FSCPTR; 8 PARA EMPEZAR
1050    DEX
1060    STX HSCROL
1070    BNE CONT
1080    LDX #8
1090  CONT; (CONTINUAR)
1100    STX FSCPTR
1110    CPX #7
1120    BEQ COARSE
1130    JMP SYSVBV
1140  COARSE
1150    LDY TCKPTR; NUMERO DE CARACTERES A DESPLAZAR
1160    DEY
1170    BNE SCORSE; REPETIR EL CICLO HASTA QUE SE HAYA
        DESPLAZADO LA LINEA COMPLETA
1180    LDY #40
1190    JSR TCKSET; REINICIAR LA LINEA DEL CRONOMETRO
1200  SCORSE ;HACER DESPLAZAMIENTO GRUESO
1210    STY TCKPTR
1220    INC SCROLN; BYTE BAJO DE LA DIRECCION
1230    BNE RETURN
1240    INC SCROLN+1; BYTE ALTO DE LA DIRECCION
1250  RETURN
1260    JMP SYSVBV
1270  ;
1280  TCKSET
1290    LDA #LINE2&255
1300    STA SCROLN
1310    LDA #LINE2/256
1320    STA SCROLN+1
1330  ENDIT
1340    RTS
```

Tan pronto como haya digitado este programa, asegúrese de guardarlo en un disco de inmediato. Entonces, puede ejecutarlo, depurarlo si es necesario, y guardarlo nuevamente. ¡Una vez que lo vea funcionando correctamente, estará de acuerdo conmigo en que es una pantalla muy agradable, y sin duda un ejemplo de desplazamiento muy suave! Ahora, la explicación de cómo funciona el programa.

Cómo implementar Desplazamiento Fino

La razón por la que funciona tan bien el desplazamiento fino es porque tiene ocho veces la resolución del desplazamiento grueso. Cuando en un programa se usa desplazamiento grueso, éste hace que las líneas de texto salten por la pantalla (hacia arriba o hacia abajo)

un carácter completo a la vez. Pero cuando se utiliza desplazamiento fino, el texto se puede mover por la pantalla una octava parte de un carácter a la vez. He aquí cómo funciona: Mire de cerca un carácter de texto en su pantalla de video, y verá que está compuesto por una matriz de puntos diminutos. Si usa una lupa, podrá ver que hay exactamente 64 puntos en cada carácter. Cada carácter en la pantalla tiene ocho filas de puntos (o líneas horizontales) de altura, y ocho filas de puntos (o relojes de color) de ancho. Y estas filas de puntos, líneas horizontales y relojes de color, componen los incrementos utilizados en el desplazamiento fino.

Para crear e implementar una rutina de desplazamiento fino se requieren varios pasos. En primer lugar, debe ir a su lista de despliegue y habilitar el desplazamiento fino activando algunos bits de la instrucción LMS que aparece antes de cada línea que se desea desplazar. Cuando se ha activado el bit 4 de la instrucción LMS, la línea que sigue a la instrucción LMS podrá ser desplazada horizontalmente. Cuando se haya activado el bit 5 de la instrucción LMS, la línea que sigue a la instrucción se podrá desplazar verticalmente. Si se activan los bits 4 y 5 de una instrucción LMS, entonces la línea que sigue a la instrucción se podrá desplazar horizontal y verticalmente.

Échele un vistazo a las líneas de la 500 a la 520 del programa que acaba de digitar, y verá que la instrucción LMS antes de la línea cuya etiqueta es SCROLN (la línea que se desplaza) es $57. Mire la versión anterior de su programa, la del capítulo 13, en la que el desplazamiento fino no estaba habilitado. Verá que esta instrucción LMS se ha cambiado de $47 a $57.

El número $47 expresado en notación binaria corresponde a 0100 0111. Como cualquier programador de lenguaje ensamblador podrá ver claramente, el bit 4 de dicho número binario (el quinto bit desde la derecha, dado que el primer bit de un número binario es el bit 0), es cero. En otras palabras, el bit 0 no ha sido activado. Cuando el bit 4 es activado, el número se convierte en 0101 0111, o $57. Por lo tanto, se activa el desplazamiento fino de la línea cuya etiqueta es SCROLN, y las líneas de la 500 a la 520 de nuestro programa se convierten en:

```
500     .BYTE $70,$70,$70,$70,$57
510   SCROLN; (ESTA ES LA LINEA QUE DESPLAZAREMOS)
520     .WORD $00; (UN ESPACIO QUE SERA LLENADO DESPUES)
```

Ahora supongamos que quiere desplazar SCROLN de manera vertical en lugar de horizontal. ¿Qué haría usted? Bueno, tan sólo debe activar el bit 5 de la instrucción LMS en la línea 500. Entonces, el $57 que ve en esa línea se convertirá en $67 o, en notación binaria, 0110 0111. Si quiere habilitar el desplazamiento horizontal y vertical de la línea, tan sólo debe cambiar la instrucción LMS a $77 (0111 0111).

El desplazamiento fino, al igual que el desplazamiento grueso, se puede realizar sobre cualquier número de líneas de texto en la pantalla. Sólo hay que establecer el bit correcto (o bits) en la instrucción LMS adecuada (o instrucciones), y el tipo deseado de

desplazamiento puede ser aplicado a cada línea seleccionada. Sin embargo, puede preguntarse: ¿Y si no hay ninguna instrucción LMS para una línea que desea desplazarse? Bueno, en ese caso, simplemente puede escribir una. No hay absolutamente ninguna razón para que una lista de despliegue no pueda tener una instrucción LMS en cada línea de la pantalla. Si desea desplazar una pantalla entera, debe poner, de hecho, una instrucción LMS delante de cada línea. Hasta ahora hemos hablado acerca de habilitar el desplazamiento fino. Pero ahora que sabe cómo habilitarlo, ¿cómo se hace en realidad?

Buena pregunta.

Cuando se activa el desplazamiento fino de una línea, se entrega el control de la línea a uno de los dos registros de desplazamiento que residen en el sistema operativo de su Atari. Si ha autorizado el desplazamiento horizontal en una línea determinada de la pantalla, entonces esa línea se convierte en un objeto controlado por un registro de desplazamiento horizontal (horizontal scroll register), el cual es abreviado como HSCROL y que está situado en la dirección de memoria $D404. Cuando se ha activado un desplazamiento vertical para una línea determinada de la lista de despliegue, entonces la línea será controlada por un registro de desplazamiento vertical (vertical scroll register), o VSCROL, situado en la dirección $D405. Si se activan tanto el desplazamiento horizontal como el vertical, entonces la línea será controlada tanto por el registro HSCROL y como por el registro VSCROL. Una vez que el control de la línea ha sido entregado a HSCROL, VSCROL o a ambos, entonces puede activar el desplazamiento fino simplemente cargando un valor en el registro (o registros) de desplazamiento adecuado. Cuando se carga un número en el registro HSCROL, cada línea de la lista de despliegue que ha sido puesta bajo el control de dicho registro se desplazará a la derecha el número de relojes de color que hayan sido cargados en el registro VSCROL, y cada línea a la que se le ha activado el desplazamiento vertical se desplazará hacia arriba el número de líneas horizontales que se ha especificado.

Combinando el desplazamiento fino con el grueso

Sin embargo, hay un problema. Los registros de desplazamiento de su computador son registros de 8 bits, y sólo cuatro de los 8 bits en cada registro se utilizan alguna vez. Esto significa que el desplazamiento fino tiene esta limitación. Para funcionar correctamente, el desplazamiento fino debe ser combinado con el desplazamiento grueso, el que puede desplazar tantos datos como se puedan programar. En general, la mejor forma de combinar el desplazamiento fino con el desplazamiento grueso es realizar el desplazamiento fino a una línea o columna de caracteres por siete relojes de color o líneas horizontales y a continuación restablecer el correspondiente registro de desplazamiento a su valor inicial y luego implementar un desplazamiento grueso. Haga un ciclo a través de este tipo de procedimiento una y otra vez, y el resultado será una desplazamiento fino y suave. Puede ver cómo funciona este procedimiento estudiando y

experimentando con la rutina de desplazamiento fino del programa de pantalla de títulos que hemos estado examinando.

Suavizando la acción de su desplazamiento

Eso es todo lo que hay que saber con respecto al desplazamiento fino, si no le importa tener saltos bruscos o movimientos irregulares de vez en cuando en la pantalla de video. Si este tipo de desórdenes no le apetece, y estoy seguro que así es, entonces podemos seguir adelante y hablar acerca de cómo hacer una operación de desplazamiento fino perfecto: suave, libre de saltos bruscos y movimientos irregulares.

Como recordará haber visto en el capítulo anterior, la imagen del monitor de su computador es dibujada por un cañón de electrones 60 veces cada segundo, y entre cada refresco de pantalla hay una fracción de segundo en que sucede un apagón total de la pantalla y que tiene lugar demasiado rápido para que usted lo pueda ver. Bueno, al escribir una rutina de desplazamiento fino en lenguaje ensamblador y no tomar algunas precauciones especiales, la pantalla puede (de hecho, casi siempre) ensuciarse un poco y saltar de vez en cuando. Esto es porque algunas de las acciones de desplazamiento que usted ha programado a veces se ejecutarán mientras la pantalla es dibujada por el cañón de electrones que se encuentra dentro del tubo de su monitor de video. Pero hay una manera de evitar que esto suceda. La gente que diseñó su computador Atari le ha entregado algo que se llama "vector de interrupción del borrado vertical - Vertical Blank Interrupt (VBI) vector", y una vez que aprender a utilizar este vector, podrá realizar todo tipo de trucos gráficos, en tiempo real y sin peligro alguno de estropear la pantalla de su computador.

Un vector, como usted sabe, es un punto en el sistema operativo de su computador que contiene la dirección de una rutina específica. El propósito principal de un vector es darle a usted un método sencillo para la aplicación de una rutina de uso frecuente. Cuando salta a un vector del sistema operativo en el transcurso de un programa, el programa automáticamente saltará a la rutina del sistema operativo a la que el vector apunta, y usted podrá utilizar esta rutina sin tener que escribir desde cero todo el código de la misma. También los vectores a veces pueden ser utilizados de otra manera. A veces se puede "robar" un vector, es decir, se puede cambiar su valor de modo que apunte a una rutina que has escrito usted mismo, en vez de apuntar a la rutina original del sistema operativo. Eso significa que a veces se puede utilizar un vector como un método fácil para controlar el comportamiento del sistema operativo de su computador. Y eso nos lleva al siguiente punto: vectores de interrupción de borrado vertical.

En realidad, hay dos vectores VBI en su computador, y cada uno tiene su correspondiente puntero en el sistema operativo de su Atari. Uno de estos vectores se llama VVBLKI ("I" de "inmediato"), y el puntero que se puede utilizar para acceder a ella se encuentra en la dirección de memoria $0222. El puntero del otro VBI se llama VVBLKD ("D" de "diferido (deferred)") y reside en la dirección de memoria $0224.

Cada vez que su Atari comienza una interrupción de borrado vertical ("vertical blank interrupt - VBI"), le echa una mirada al contenido del puntero VVBLKI. Si el programa que se está procesando no usa el puntero VVBLKI, entonces dicho puntero contienen nada más que una instrucción para ir a una dirección de memoria predeterminada: en concreto, la dirección de memoria $ E4F (a la que Atari le ha puesto la etiqueta SYSVBV). La dirección de memoria $E45F en general no contiene nada interesante. Lo que contiene normalmente es una instrucción para que su equipo continúe su procesamiento normal. Sin embargo, al "robar" el vector VVBLKI puede hacer que apunte a cualquier rutina que quiera, por lo general una que usted mismo ha escrito. Luego, 60 veces por segundo (cada vez que su computador comienza su interrupción de borrado vertical a 60 Hertz) procesará automáticamente la rutina cuya dirección se ha almacenado en el puntero VVBLKI. Cuando su rutina termine, el equipo reanudará su procesamiento normal.

El otro vector VBI de su equipo, VVBLKD, también apunta directamente a un punto de salida a menos que haya sido "robado" por algún programa. El punto de salida normal del vector VVBLKD es la dirección de memoria $E462, a la que Atari le ha puesto la etiqueta XITVBV, y funciona exactamente igual que SYSVBV. Simplemente termina el período de la interrupción de borrado vertical de su computador y le permite a éste reanudar su proceso normal. Una vez que entienda cómo funcionan las interrupciones VVBLKI y VVBLKD, no será difícil "robarlas". Aquí está todo lo que tiene que hacer:

- Escriba una rutina que le gustaría ejecutar durante el transcurso de una interrupción de borrado vertical.
- Asegúrese de que la rutina termina con un salto a SYSVBV o a XITVBV (dependiendo de si el vector que se utiliza es inmediato o diferido).
- Guarde la dirección de su rutina en VVBLKI para una interrupción inmediata, o en VVBLKD para una interrupción diferida.

Una vez que se toman esas medidas, su computador procesará su nueva rutina 60 veces cada segundo, justo antes de que comience cada interrupción VBI, si es que ha utilizado el vector inmediato, o justo antes de que regrese de cada interrupción VBI, si ha utilizado el vector diferido. Eso hace que el robo de vector sea una técnica muy útil para escribir programas que requieren una alta velocidad y gran rendimiento para su procesamiento, tales como el procesamiento de las rutinas que tienen que ver con gráficos y sonido.

En este punto debe preguntarse por qué hay dos vectores que puede robar, y cuáles son sus diferencias.

Bueno, la razón es simplemente porque ciertas rutinas escritas por el usuario se ejecutan mejor al inicio del período de borrado vertical, mientras que otras no se debe ejecutar hasta que ha terminado la VBI. Más información sobre este punto se puede encontrar en el libro *De Re de Atari y en el documento Atari 400/800 Technical Reference Notes*.

Una cosa más. . .

Sólo hay un hecho importante que debe recordar acerca del robo de vectores VBI. Después de haber robado un vector, hay una pequeña probabilidad de que se inicie una interrupción justo después de que el primer byte del puntero que está utilizando haya sido actualizado, pero antes de que se haya cambiado el segundo byte. Si esto ocurre, su programa podría producir un error. Pero esta posibilidad se puede evitar fácilmente. Todo lo que tiene que hacer es utilizar una rutina del sistema operativo que ha escrito la gente que diseñó su computador. Esta rutina se llama SETVBV, y comienza en la dirección de memoria $E45C.

He aquí cómo se usa la rutina SETVBV: En primer lugar, se debe cargar el registro Y del 6502 con el byte bajo de la dirección de la rutina que le indica al computador que comience una rutina de cambio de vector. A continuación, se debe cargar el registro X con el byte alto de la dirección. Luego tiene que cargar el acumulador con un 6 para una VBI inmediata o un 7 para una VBI diferida. Luego se debe hacer un JSR SETVBV, y su interrupción se habilitará de manera segura. Esto es todo lo que debe saber acerca de cómo utilizar en borrado vertical en programas en lenguaje ensamblador para desplazamiento fino.

Personalización del juego de caracteres

Su computador Atari tiene un juego de caracteres integrado muy bueno. Si tiene uno de los últimos modelos de Atari, puede tener incluso dos juegos de caracteres incorporados en su ROM. Pero, ¿Le gustaría ser capaz de crear sus propios juegos de caracteres, incluyendo no sólo letras, números y símbolos especiales de texto, sino caracteres gráficos también?



Bueno, puede hacerlo muy fácilmente si sabe lenguaje ensamblador. Puede hacerlo a la velocidad del rayo, también (y no a paso de caracol como cuando el BASIC tiene que realizar la modificación de caracteres). En realidad es bastante fácil de diseñar un juego de caracteres en un computador Atari. Como he señalado anteriormente en este capítulo, cada carácter que imprime su computador en su monitor de video se compone de una matriz de 8 por 8 puntos. Esta grilla de 8 por 8 se almacena en la memoria RAM de su equipo en la forma de ocho bytes de datos. La letra A, por ejemplo, se almacena en la memoria de su computador como una cadena de dígitos binarios que pueden ser representados de la siguiente manera:

Notación Binaria	**Notación Hexadecimal**	**Apariencia**
0000 0000	00	
0001 1000	18	
0011 1100	3C	
0110 0110	66	
0110 0110	66	
0111 1110	7E	
0110 0110	66	
0000 0000	00	

El conjunto de caracteres primario de su computador se compone de 128 letras, cada una compuesta de 64 puntos que podrían ser dispuestos en el mismo formato de 8 por 8 de la letra "A". De hecho, ese es precisamente el formato en el que las letras son dispuestas cuando aparecen en la pantalla de su computador. Dado que hay 128 caracteres en un juego, y puesto que cada carácter se compone de ocho bytes de datos, un juego completo de caracteres ocupa 1.024 bytes de memoria RAM. Ponga todos los datos juntos, y tendrá

una tabla muy larga. También la tabla debe comenzar en un límite de 1K, debido a la arquitectura de su computador. Los juegos de caracteres se pueden guardar en cualquier lugar de la memoria de su computador Atari, pero la dirección del juego de caracteres que se está utilizando actualmente siempre se almacena en un puntero específico del sistema operativo del computador. Dicho puntero, al que los ingenieros de Atari le pusieron la etiqueta CHBAS, se encuentra en la dirección de memoria $2F4.

Para localizar un carácter en su computador sólo tiene que saber dos cosas: el valor actual de CHBAS, y el código ATASCII del carácter que está buscando. Sume el número correspondiente al código ATASCII del carácter al valor de CHBASE, y obtendrá la dirección de memoria del carácter que está buscando. Si su computador es un Atari 400 o un Atari 800, entonces su juego de caracteres está incorporado en la ROM y por lo tanto, no puede ser modificado. Si tiene un modelo de computador Atari posterior, el juego de caracteres tiene una dirección de memoria en RAM y por lo tanto puede ser accedido de manera un poco más fácil. Pero, si desea modificar el juego de caracteres integrado de su computador, entonces tiene que hacerlo de una manera más bien indirecta, no importa qué modelo de Atari posea usted.

La mejor manera de modificar un juego de caracteres es copiándolo en algún bloque libre de memoria RAM y que sea de fácil acceso. Luego puede modificar el contenido de CHBAS de modo que apunte a la dirección de inicio de su propio bloque de caracteres en lugar del conjunto de caracteres que viene integrado en su computador. Entonces podrá utilizar el juego de caracteres que quiera, ya sea el que viene incorporado en su Atari o el que ha creado usted mismo.

Una vez que haya definido un nuevo juego de caracteres y lo haya almacenado en la memoria RAM, todo lo que tiene que hacer para cambiar de un juego de caracteres a otro es cambiar el contenido de CHBAS. Esto hace que sea fácil escribir rutinas de animación de caracteres. Todo lo que tiene que hacer para animar un conjunto de caracteres es dibujar varios juegos de caracteres diferentes que varían ligeramente, y luego debe alternar entre ellos, simplemente cambiando el contenido del puntero CHBAS. BASIC es demasiado lento para manejar muy bien un trabajo como éste, pero cuando se conoce el lenguaje ensamblador, se puede animar juegos de caracteres en tiempo real, a velocidades rápidas como el rayo. En un momento le mostraré un programa que puede utilizar para diseñar sus propios juegos de caracteres. El programa tiene dos partes bien diferenciadas. La primera parte copia la tabla completa de caracteres Atari a un lugar en la memoria seleccionada por el programador. La segunda parte altera un sólo carácter - la letra "A". ¡Sin embargo, la misma técnica puede ser usada para alterar cualquier otro carácter o, si lo desea, todos los caracteres!

Me gustaría hacer dos observaciones más antes de que escriba el siguiente programa. En primer lugar, me gustaría señalar que la primera parte del programa, la parte que mueve el conjunto de caracteres, puede utilizarse para mover cualquier bloque de datos desde cualquier lugar de la RAM a cualquier otro lugar en la memoria RAM de su computador.

Es una herramienta útil, ya que a menudo en los programas en lenguaje ensamblador los datos tienen que ser trasladados de un bloque de memoria a otro. La parte en que se mueven los datos en el programa que sigue es especialmente buena, ya que fue diseñada para mover bloques de memoria por páginas enteras a la vez, usando punteros de 8 bits en lugar de punteros de 16 bits, ahorrando una cantidad considerable de tiempo de procesamiento. El segundo punto que me gustaría notar es que hay maneras de crear juegos de caracteres sin tener que pasar por la monotonía de diseñar los caracteres dibujándolos sobre papel milimetrado y después escribiéndolos en la memoria un byte a la vez. Hay programas excelentes en el mercado que pueden ayudarle a crear sus propios juegos de caracteres directo en la pantalla de su computador Atari, usando un cursor, un conjunto de menús y comandos de teclado. Así que si está interesado en los gráficos por computador, podría ser ventajoso para usted que le echara un vistazo a algunos programas generadores de caracteres de calidad profesional.

Pero esto es suficiente de mi parte. He aquí su programa:

MOVIENDO Y MODIFICANDO UN JUEGO DE CARACTERES

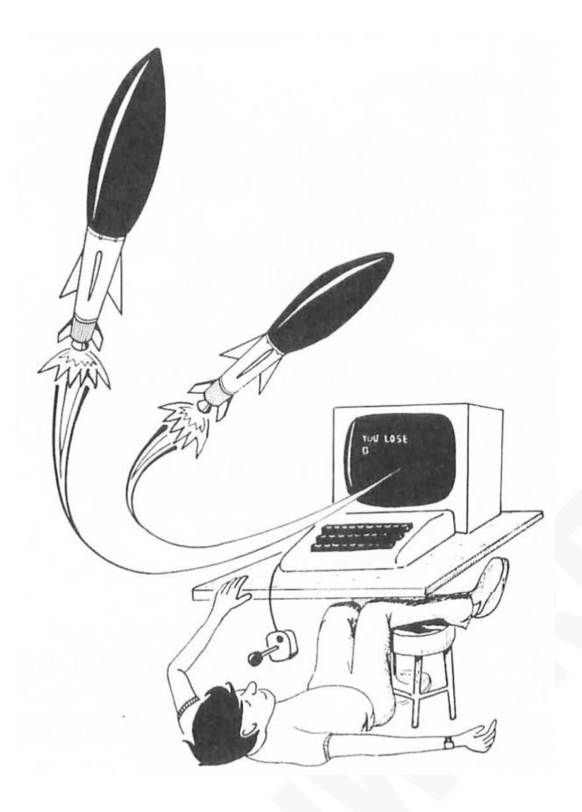
```
10     ;
20     ;MODIFICANDO CARACTERES GRAFICOS
30     ;
50       *=$0600
60       JMP MOVDAT
70     ;
80     CHBAS=$02F4
90     NEWADR=$5000
100    TABLEN=1024
110    ;
120    MVSRCE=$B0; PUNTERO A LA PAGINA CERO
130    MVDEST=MVSRCE+2; IDEM
140    CHRADR=MVDEST+2; IDEM
150    ;
160    LENPTR=$6000; OTRO PUNTERO
170    RAMCHR=LENPTR+2; IDEM
180    ;
190    SHAPE .BYTE $18,$DB,$42,$7E,$18,$7E,$66,$E7;UN HOMBRE
200    ;
210    START=MOVDAT
220    ;
230    MOVDAT
240    ;
250    ;ALMACENAR VALORES EN PUNTEROS
260    ;
270      LDA #0
280      STA MVSRCE; BYTE BAJO
290      LDA CHBAS; BYTE ALTO
300      STA MVSRCE+1; BYTE ALTO
310    ;
320      LDA #NEWADR&255
```

```
330      STA MVDEST
340      LDA #NEWADR/256
350      STA MVDEST+1
360    ;
370      LDA #TABLEN&255
380      STA LENPTR
390      LDA #TABLEN/256
400      STA LENPTR+1
410    ;
420    ;START MOVE
430    ;
440      LDY #0
450      LDX LENPTR+1
460      BEQ MVPART
470    MVPAGE
480      LDA (MVSRCE),Y
490      STA (MVDEST),Y
500      INY
510      BNE MVPAGE
520      INC MVSRCE+1
530      INC MVDEST+1
540      DEX
550      BNE MVPAGE
560    MVPART
570      LDX LENPTR
580      BEQ MVEXIT
590    MVLAST
600      LDA (MVSRCE),Y
610      STA (MVDEST),Y
620      INY
630      DEX
640      BNE MVLAST
650      MVEXIT
660    ;
670    ;PARTE II: CAMBIAR CARACTER
680    ;
690    ;CAMBIAREMOS EL CARACTER "A"
700    ;
710      LDA #33;CODIGO RAM: "A"
720      STA RAMCHR
730    ;
740    ;AHORA CALCULAMOS LA DIRECCION DE RAMCHR
750    ;
760      LDA #0
770      STA RAMCHR+1; LO LIMPIAMOS
780      LDA RAMCHR; #33:UNA "A"
790      CLC
800      ASL A; MULTIPLICAMOS POR 2
810      ROL RAMCHR+1; OBTENER ACARREO
820      ASL A; OTRA VEZ
830      ROL RAMCHR+1
840      ASL A; Y OTRA VEZ
```

```
850       ROL RAMCHR+1
860       STA RAMCHR; MULTIPLICAMOS POR 8 Y LISTO
870     ;
880     FNDADR
890       CLC
900       LDA RAMCHR
910       ADC #NEWADR&255
920       STA CHRADR
930       LDA RAMCHR+1
940       ADC #NEWADR/256
950       STA CHRADR+1
960     ;
970     ;AHORA CAMBIAMOS EL CARACTER
980     ;
990     NEWCHR
1000      LDY #0; NUMERO DE BYTES+1
1010    DOSHAPE
1020      LDA SHAPE,Y
1030      STA (CHRADR),Y
1040      INY
1050      CPY #8
1060      BCC DOSHAPE; REPETIR HASTA TERMINAR
1070    ;
1080    ;ALMACENAR LA DIRECCION DEL NUEVO JUEGO DE CARACTERES EN
        CHBAS
1090    ;
1100      LDA #NEWADR/256; BYTE ALTO
1110      STA CHBAS
1120    ;
1130    IFIN
1140      RTS
```

Gráficos Player-Missile

Aunque la animación puede ser programada alternando juegos de caracteres, una manera mucho más fácil de animar caracteres en lenguaje ensamblador es sacar ventaja de una característica gráfica especial de los computadores Atari llamada "gráficos player-missile". Los gráficos player-missile es una técnica para programar animaciones utilizando caracteres gráficos llamados (no es de extrañar) players y missiles. Su computador Atari está equipado con cuatro players, numerados del 0 al 3, y cuatro missiles, uno para cada player. Si escribe un programa que no requiere missiles, se pueden combinar los cuatro missiles para formar un quinto player. Los players y los missiles pueden ser utilizados en cualquier modo gráfico, y se pueden dibujar y mover de manera totalmente independiente de los otros objetos de la pantalla. Pueden pasar por encima o debajo de otros objetos de la pantalla, y por encima o por debajo de otros players o missiles. ¡Alternativamente, pueden ser programados para que se detengan o incluso exploten cuando impacten otros objetos en la pantalla u otro player o missile!



Cada uno de los cuatro players de su Atari tiene 8 bits de ancho, tal como un carácter gráfico ordinario. Sin embargo, los players pueden ser mucho más altos que anchos. Su altura máxima es de 128 o 256 bytes de alto, dependiendo de su resolución vertical. Esto significa que un player puede ser tan alto como la altura de la pantalla de video. Cada player tiene su propio registro de color. Así que al usar players puede agregar a un programa más color del que normalmente tiene disponible. Los players suelen ser entidades de un solo color. Sin embargo, es posible fusionar dos o más players para formar un player de múltiples colores poniendo un player encima de otro.

Los players pueden tener una resolución vertical de una o dos líneas horizontales (scan lines). La resolución horizontal máxima de un player es de ocho pixeles, pero el ancho horizontal de cada píxel es variable. Cada píxel puede tener 8 relojes de color, 16 relojes de color o 32 relojes de color de ancho. Esta elección depende del programador. Los missiles utilizados en los gráficos player-missiles son puntos de luz de 2 bits de ancho que pueden ser utilizados como balas, estrellas, u otros objetos gráficos pequeños. O, como se mencionó anteriormente, se pueden combinar para formar un quinto player de tamaño completo.

Los players se componen de una matriz de puntos, tal como los caracteres de texto estándar. Por lo tanto, pueden ser diseñados en papel cuadriculado, de la misma manera que se crean los caracteres de texto comunes. Sin embargo, antes de comenzar a diseñar un player en papel cuadriculado, sería una buena idea recordar cómo se ve en la pantalla un player cuando se encienden todos sus bits. Cuando un player está completamente relleno de bits, se ve como una cinta que se extiende desde la parte superior a la parte

inferior de la pantalla de video. En la mayoría de sus programas no necesitará toda la altura disponible que tienen los players. Antes de empezar a dibujar a un player, por lo general es una buena idea "borrar" la "cinta" completa rellenándola con ceros. Así la cinta entera se volverá invisible. Entonces puede dibujar su player de la misma forma en que dibuja un carácter de texto convencional, rellenando su forma con bits "encendidos", o unos binarios. Cuando haya terminado de dibujar los players de esta manera, podrá almacenarlos en casi cualquier lugar en la memoria RAM. Puede decirle a su equipo dónde se encuentran los players y los missiles simplemente almacenando la dirección donde comienzan en la RAM en el puntero del sistema operativo llamado PMBASE. La dirección de PMBASE es de $D407.

Dado que los players y los missiles son objetos gráficos, se considera una buena práctica de programación el almacenarlos en la memoria RAM alta, justo debajo de donde se almacena la pantalla de su ordenador. Los programadores de Atari, que generalmente saben de lo que están hablando, recomiendan que almacene la RAM del despliegue de sus players y missiles 2 kilobytes por debajo de la parte superior de la memoria del computador de Atari. Una vez que deduzca dónde almacenará la RAM de los players y missiles, y en consecuencia, a qué dirección señalará PMBASE, puede comenzar a almacenar los datos de sus players y missiles directo en la RAM. Aquí hay una tabla que muestra dónde comienza la RAM de cada player y missile, con respecto a la dirección almacenada en PMBASE:

RESOLUCION DE DOS LINEAS

DESPLAZAMIENTO DESDE PMBASE	**CONTENIDO**
+ 0 - 383	Sin uso
+ 384 - 511	Missiles
+ 512 - 639	Player 0
+ 640 - 767	Player 1
+ 768 - 895	Player 2
+ 896 - 1023	Player 3
+1023	Fin de la RAM para los P/M

Debe comenzar dentro el límite de direcciones de 1K.

RESOLUCION DE UNA SOLA LINEA

DESPLAZAMIENTO DESDE PMBASE	**CONTENIDO**
+ 0 - 767	Sin uso
+ 768 - 1023	Missiles
+ 1024 - 1279	Player 0
+ 1280 - 1535	Player 1
+ 1536 - 1791	Player 2
+ 1792 - 2047	Player 3
+ 2046	Fin de la RAM para los P/M

Debe comenzar dentro el límite de direcciones de 2K.

Cuando los players y missiles son dibujados y almacenados en la memoria RAM, es bastante simple moverlos por la pantalla. En el sistema operativo de su computador hay un conjunto de registros de memoria que se utilizan para hacer un seguimiento de la posición horizontal de todos los players y missiles. Estos registros tienen etiquetas que van desde HPOSP0 hasta HPOSP3 (para los players) y desde HPOSM0 hasta HPOSM3 (para los missiles). Las direcciones de estos registros son:

REGISTRO HORIZONTAL DE:	ETIQUETA	DIRECCIÓN
Player 0	HPOSP0	D000
Player 1	HPOSP1	D001
Player 2	HPOSP2	D002
Player 3	HPOSP3	D003
Missile 0	HPOSM0	D004
Missile 1	HPOSM1	D005
Missile 2	HPOSM2	D006
Missile 3	HPOSM3	D007

Cuando desee mover un player o un misil desde una posición horizontal a otra, todo lo que tiene que hacer es cambiar el valor del registro horizontal apropiado.

Cambiar la posición vertical de un player o un misil es un poco más complicado. Para mover un player hacia arriba o hacia abajo, hay que "borrar" el player de la cinta vertical en la que aparece rellenando ese espacio con ceros. Luego tiene que volver a dibujarlo en una posición superior o inferior dentro del bloque de memoria RAM que constituye la cinta.

El programa de más adelante, el último programa de este libro, es una demostración que muestra cómo utilizar los gráficos player-missile. Como un bono especial, le mostrará también cómo usar el joystick en programas en lenguaje ensamblador.

La parte del programa que lee la posición del joystick comienza con un conjunto de instrucciones que activan el bit 2 de un registro llamado PACTL (de "Port A ConTroL - Control del Puerto A") situado en la posición de memoria $D302. Cuando el bit 2 de PACTL se activa, se habilita la lectura de los controles de juego. Entonces se pueden leer los interruptores de cada una de las posiciones del joystick conectado al puerto de la misma forma como se leen en los programas BASIC. Cuando escriba este programa y lo ejecute, será capaz de mover por la pantalla un pequeño corazón rosa usando un joystick. Cuando el corazón desaparezca al llegar al borde de la pantalla, siga empujando el control, y verá que vuelve a aparecer en el borde opuesto de la pantalla, justo donde lo espera, moviéndose en la misma dirección.

Este programa utiliza interrupciones de borrado vertical (vertical blank interrupts), al igual que la rutina de desplazamiento suave presentada al comienzo de este capítulo. Como verá cuando escriba el programa y lo ejecute, el lenguaje ensamblador es el mejor lenguaje que se puede utilizar cuando se trabaja con gráficos player/missiles. Si alguna vez has trabajado con gráficos player/missiles utilizando BASIC, pronto verá que la acción es mucho más rápida y mucho más suave cuando se programa en lenguaje ensamblador.

```
10     ;
20     ;RUTINA DE GRAFICOS PLAYER-MISSILE
```

```
30    ;
50      *=$5000
60      JMP START
70    ;
80    RAMTOP=$6A ;PUNTERO DE LA PARTE ALTA DE LA RAM
90    VVBLKD=$0224 ;RUTINA DE INTERRUPCION
100   SDMCTL=$022F ;SOMBRA DEL CONTROL DE DMA
110   SDLSTL=$0230 ;SDLST, BYTE BAJO
120   STICK0=$0278
130   PCOLR0=$02C0 ;COLOR DEL PLAYER
140   COLOR2=$02C6 ;COLOR DEL FONDO
150   ;
160   HPOSP0=$D000 ;POSICION HORIZONTAL DEL PLAYER
170   GRACTL=$D01D
180   PACTL=$D302 ;CONTROL DEL PUERTO DE JOYSTICK
190   PMBASE=$D407 ;DIRECCION BASE DEL PLAYER MISSILE
200   SETVBV=$E45C ;ACTIVA INTERRUPCION
210   XITVBV =$E462 ;SALIR DE LA INTERRUPCION
220   ;
230   HRZPTR=$0600 ;PUNTERO CON LA POSICION HORIZONTAL
240   VRTPTR=HRZPTR+1 ;PUNTERO CON LA POSICION VERTICAL
250   OURBAS=VRTPTR+1 ;NUESTRO PMBASE
260   TABSIZ=OURBAS+2 ;TAMAÑO DE LA TABLA
270   FILVAL=TABSIZ+2 ;VALOR BLKFIL
280   ;
290   TABPTR=$B0 ;PUNTERO DE LA DIRECCION DE LA TABLA
300   TABADR=TABPTR+2 ;DIRECCION DE LA TABLA
310   ;
320   PLBOFS=512 ;DESPLAZAMIENTO DE LA BASE DEL PLAYER
330   PLTOFS=640 ;DESPLAZAMIENTO DE LA PARTE ALTA DEL PLAYER
340   ;
350   SHAPE .BYTE $00,$6C,$FE,$FE,$7C,$38,$10,$00
360   ;
370     START
380   ;
390   ;LIMPIAR PANTALLA
400   ;
410     LDA #0
420     STA FILVAL
430     LDA SDLSTL
440     STA TABPTR
450     LDA SDLSTL+1
460     STA TABPTR+1
470     LDA #960&255 ;BYTES POR PANTALLA
480     STA TABSIZ
490     LDA #960/256
500     STA TABSIZ+1
510     JSR BLKFIL
520   ;
530   ;DEFINIR VARIABLES DE GRAFICOS PLAYER-MISSILE
540   ;
550     LDA #0
```

```
560      STA COLOR2 ;FONDO NEGRO
570      LDA #$58
580      STA PCOLR0 ;PLAYER ROSA
590    ;
600      LDA #100 ;ACTIVAR POSICION HORIZONTAL
610      STA HRZPTR
620      STA HPOSP0
630    ;
640      LDA #48 ;ACTIVA POSICION VERTICAL
650      STA VRTPTR
660    ;
670      LDA #0 ;LIMPIA OURBASE
680      STA OURBAS
690      STA OURBAS+1
700    ;
710      SEC
720      LDA RAMTOP
730      SBC #8
740      STA PMBASE ;BASE=RAMTOP-2K
750      STA OURBAS+1 ;GUARDAR DIRECCION BASE
760    ;
770      LDA #46
780      STA SDMCTL ;ACTIVAR DMA DE PLAYER-MISSILE
790    ;
800      LDA #3
810      STA GRACTL ;ACTIVAR DESPLIEGUE DE PLAYER-MISSILE
820    ;
830    ;RELLENA LA RAM DE LOS PLAYER-MISSILE CON CEROS PARA
       LIMPIARLA
840    ;
850      CLC
860      LDA OURBAS
870      ADC #PLBOFS&255
880      STA TABADR
890      STA TABPTR
900      LDA OURBAS+1
910      ADC #PLBOFS/256
920      STA TABADR+1
930      STA TABPTR+1
940    ;
950      SEC
960      LDA #PLTOFS&255
970      SBC #PLBOFS&255
980      STA TABSIZ
990      LDA #PLTOFS/256
1000     SBC #PLBOFS/256
1010     STA TABSIZ+1
1020   ;
1030     LDA #0
1040     STA FILVAL
1050     JSR BLKFIL
1060   ;
```

```
1070   ;DEFINIR PLAYER
1080   ;
1090     PLAYER
1100   ;
1110   ;DIBUJAR PLAYER
1120   ;
1130     JSR DRAWPL
1140   ;
1150   ;ACTIVAR INTERRUPCION
1160   ;
1170     LDY #INTRPT&255
1180     LDX #INTRPT/256
1190     LDA #7
1200     JSR SETVBV
1210   ;
1220   INTRPT
1230     LDA #RDSTIK&255
1240     STA VVBLKD
1250     LDA #RDSTIK/256
1260     STA VVBLKD+1
1270   ;
1280   ;CICLO INFINITO
1290   ;
1300   INFIN
1310     JMP INFIN
1320   ;
1330   ;LEER JOYSTICK
1340   ;
1350   RDSTIK
1360     LDA #4
1370     ORA PACTL ;ACTIVAR BIT #5
1380   ;
1390     LDA STICK0
1400     CMP #$F ;EL JOYSTICK APUNTA HACIA ARRIBA?
1410     BEQ RETURN ;SI, NO HAY ACCION
1420   ;
1430   TRYAGN
1440     CMP #$07 ;MOVER A LA DERECHA
1450     BNE TRYAG2
1460     LDX HRZPTR
1470     INX
1480     STX HRZPTR
1490     STX HPOSP0
1500     JMP RETURN
1510   ;
1520   TRYAG2
1530     CMP #$0B ;MOVER A LA IZQUIERDA
1540     BNE TRYAG3
1550   ;
1560     LDX HRZPTR
1570     DEX
1580     STX HRZPTR
```

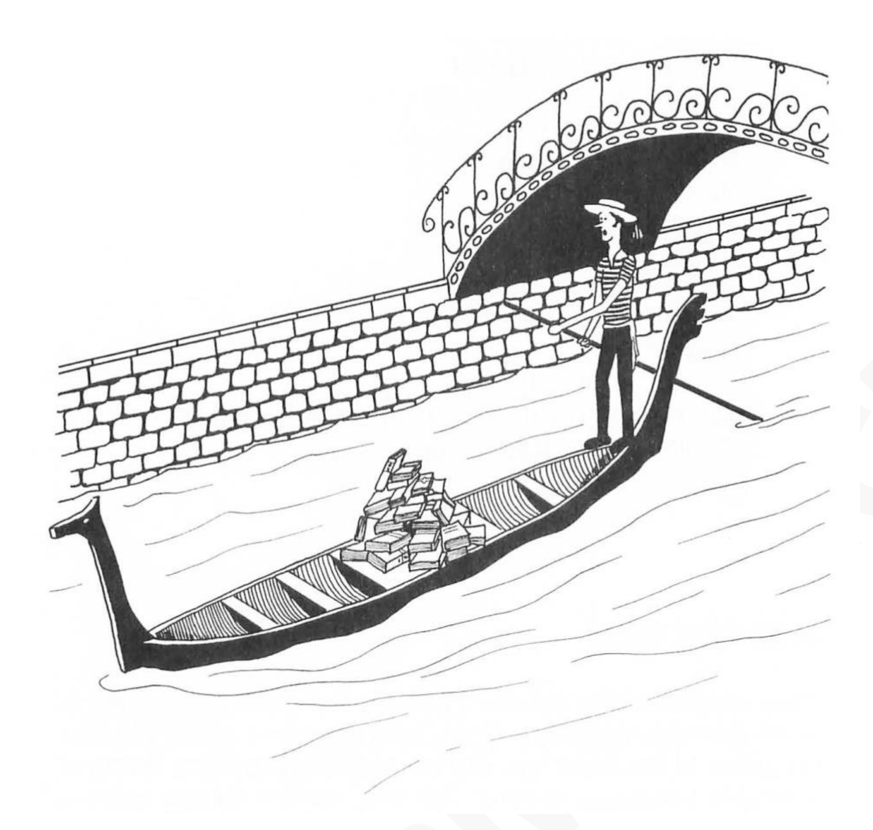
```
1590     STX HPOSP0
1600     JMP RETURN
1610   ;
1620   TRYAG3
1630     CMP #$0D ;MOVER HACIA ABAJO
1640     BNE TRYAG4
1650   ;
1660     INC VRTPTR
1670     JSR DRAWPL
1680     JMP RETURN
1690   ;
1700   TRYAG4
1710     CMP #$0E ;MOVER HACIA ARRIBA
1720     BNE RETURN
1730   ;
1740     DEC VRTPTR
1750     JSR DRAWPL
1760     JMP RETURN
1770   ;
1780   RETURN
1790     JMP XITVBV
1800   ;
1810   ;RUTINA PARA RELLENAR BLOQUE
1820   ;
1830   BLKFIL
1840   ;
1850   ;PRIMERO HACER PAGINAS COMPLETAS
1860   ;
1870     LDA FILVAL
1880     LDX TABSIZ+1
1890     BEQ PARTPG
1900     LDY #0
1910   FULLPG
1920     STA (TABPTR), Y
1930     INY
1940     BNE FULLPG
1950     INC TABPTR+1
1960     DEX
1970     BNE FULLPG
1980   ;
1990   ;HACER LA PAGINA PARCIAL QUE QUEDA
2000   ;
2010   PARTPG
2020     LDX TABSIZ
2030     BEQ FINI
2040     LDY #0
2050   ;
2060   PARTLP
2070     STA (TABPTR),Y
2080     INY
2090     DEX
2100     BNE PARTLP
```

```
2110   ;
2120   FINI
2130     RTS
2140   ;
2150   DRAWPL
2160     PHA ;GUARDAR VALOR DEL ACUMULADOR
2170     CLC
2180     LDA TABADR
2190     ADC VRTPTR
2200     STA TABPTR
2210     LDA TABADR+1
2220     ADC #0
2230     STA TABPTR+1
2240   ;
2250     LDA #0
2260     FILLPL
2270     LDA SHAPE,Y
2280     STA (TABPTR), Y
2290     INY
2300     CPY #8
2310     BCC FILLPL ;REPETIR HASTA TERMINAR
2320     PLA ;REESTABLECER EL VALOR DEL ACUMULADOR
2330     RTS
```

Este no es el fin

Con esto concluye esta guía del viajero por el fascinante mundo del lenguaje ensamblador de Atari. Si ha escrito y guardado todos los programas de este libro, ahora tiene una biblioteca bastante amplia de rutinas en lenguaje ensamblador que puede (sin duda) mejorar y utilizar en sus propios programas. Si ha asimilado el material que rodea a las rutinas de este volumen, ahora sabe casi todo lo que necesita saber para comenzar a escribir programas muy sofisticados en lenguaje ensamblador.

Tengo solamente una sugerencia más. Si está interesado en programar más en lenguaje ensamblador Atari - y ciertamente espero que así sea - entonces hay otros dos libros que definitivamente debería tener. Estos son (en caso de que aún no lo haya adivinado) el De Re de Atari y el Atari 400/800 Technical Reference Notes, ambos publicados por Atari. Con esos dos libros, y el que acaba de terminar, debería ser capaz de hacer casi cualquier cosa de ahora en adelante en lenguaje ensamblador de Atari.



Próximamente los Apéndices. 